Detection and Diagnosis of Memory Leaks in Web Applications

by

Masoomeh Rudafshani

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Memory leaks – the existence of unused memory on the heap of applications – result in low performance and may, in the worst case, cause applications to crash. The migration of application logic to the client side of modern web applications and the use of JavaScript as the main language for client-side development have made memory leaks in JavaScript an issue for web applications. Significant portions of modern web applications are executed on the client browser, with the server acting only as a data store. Client-side web applications communicate with the server asynchronously, remaining on the same web page during their lifetime. Thus, even minor memory leaks can eventually lead to excessive memory usage, negatively affecting user-perceived response time and possibly causing page crashes. This thesis demonstrates the existence of memory leaks in the client side of large and popular web applications, and develops prototype tools to solve this problem.

The first approach taken to address memory leaks in web applications is to detect, diagnose, and fix them during application development. This approach prevents such leaks from happening by finding and removing their causes. To achieve this goal, this thesis introduces LeakSpot, a tool that creates a runtime heap model of JavaScript applications by modifying web-application code in a browser-agnostic way to record object allocations, accesses, and references created on objects. LeakSpot reports the locations of the code that are allocating leaked objects, i.e., leaky allocation sites. It also identifies accumulation sites, which are the points in the program where references are created on objects but are not removed, e.g., the points where objects are added to a data structure but are not removed. To facilitate debugging and fixing the code, LeakSpot narrows down the space that must be searched for finding the cause of the leaks in two ways: First, it refines the list of leaky allocation sites and reports those allocation sites that are the main cause of the leaks. In addition, for every leaked object, LeakSpot reports all the locations in the program that create a reference to that object. To confirm its usefulness and efficacy experimentally, LeakSpot is used to find and fix memory leaks in JavaScript benchmarks and open-source web applications. In addition, the potential causes of the leaks in large and popular web applications are identified. The performance overhead of LeakSpot in large and popular web applications is also measured, which indirectly demonstrates the scalability of LeakSpot.

The second approach taken to address memory leaks assumes memory leaks may still be present after development. This approach aims to reduce the effects of leaked memory during runtime and improve memory efficiency of web applications by removing the leaked objects or early triggering of garbage collection, Using a new tool, MemRed. MemRed automatically detects excessive use of memory during runtime and then takes actions to reduce memory usage. It detects the excessive use of memory by tracking the size of all objects on the heap. If an error is detected, MemRed applies recovery actions to reduce the overall size of the heap and hide the effects of excessive memory usage from users. MemRed is implemented as an extension for the Chrome browser. Evaluation demonstrates the effectiveness of MemRed in reducing memory usage of web applications.

In summary, the first tool provided in this thesis, LeakSpot, can be used by developers in finding and fixing memory leaks in JavaScript Applications. Using both tools improves the experience of web-application users.

Acknowledgements

First and foremost I would like to express my gratitude to my supervisor, Paul Ward. He provided me the opportunity to work on the topic of my interest and advised me how to challenge and question my work. His guidance and encouragement helped me get through many difficult times.

I would like to thank the members of my committee – Hanan Lutfiyyah, Lin Tan, Derek Rayside, and David Taylor – for their helpful suggestions and insightful comments on my work. Their valueable feedbacks improved the quality of this thesis.

The past and present members of the Shoshin group have contributed to my personal and professional growth at Waterloo. The weekly meetings were motivational and the comments that I received on my work were really helpful. I would like to thank Bernard Wong for his early comments on this work.

I would also like to thank all of my friends whom I enjoyed their companionship, and encouraged me to strive towards my goal.

I would like to give Special thanks to my family. It certainly would not have been possible for me to get this far in my studies without the hard working and sacrifices of my parents, Ghazanfar and Marzieh. Their love and encouragement were always motivating me to reach my goal. I am thankful to my big brother, Hamid, who was encouraging and helping me to pursue my interests during all years of growing up together and to my sister, Somayeh, for her endless love and support. I'm thankful to my son, Rashed, for all the joy that he brought into my life. Last but not the least, I would like express appreciation to my beloved husband, Bashir. I have been fortunate to have had him as a wonderful and supportive friend. I would also like to thank him for his insightful comments on my work.

Dedication

This thesis is dedicated to all those who spread the hope around the world.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

In a garbage-collected language such as JavaScript, a memory leak means the existence of objects on the heap of an application which are reachable from the garbage collection roots but will not be used by the program anymore. Such memory leaks do not cause significant problems for traditional web applications where pages are short-lived and most of the events on the page lead to a new page load from the server; however, the characteristics and complexity of modern web applications make memory leaks a problem [19, 39, 74]. Modern web applications, leveraging AJAX technology [104], communicate asynchronously with the server to retrieve required data; as such, they often remain on the same page for a long time without requiring a full page refresh or navigation to a new page. This situation causes the leaked memory to accumulate over time, affecting the user-perceived response time of the application [74] and possibly causing the program to fail. In the context of browsers, they affect the response time of other web applications and the whole browser system, too.

In summary, the growth in size and complexity of the client side of web applications, the potential for leaked memory accumulation due to long-lived web applications, and the consequences of memory leaks have motivated this research to focus on techniques to improve the memory efficiency of web applications by focusing on the memory leaks. The goal of this thesis is to demonstrate the extent of memory leaks in web applications and the need to solve the memory leak problem on the client side of web applications. Based on this need, the dissertation presents tool support to detect and fix memory leaks in web applications during development, and to reduce the effects of memory leaks at runtime.

These tools should help the developers of web application to reduce leakiness of applications and the users of web applications to experience memory efficient and responsive systems.

1.2 Background

This section first briefly mentions the current work on memory leak detection and discusses why those tools and techniques are not sufficient for detecting and diagnosing memory leaks in JavaScript applications. Next, it discusses the current work on error detection and recovery at runtime and explains why new solutions are needed for runtime management of memory leaks on the client side of web applications.

As a type of memory bloat, a memory leak in a managed language occurs when object references that are no longer needed are unnecessarily maintained, resulting in the existence of unused objects on the heap. Since detecting leaked memory (unused objects) is an undecidable problem [76], many of the current solutions use heuristics. Techniques based on static analysis [74] can be used to attempt the detection of such leaks; however, detection techniques based on static analysis are limited by the lack of scalable and precise reference/heap modelling (a well-known deficiency of static analysis), reflection, scalability for large programs, etc. Thus, in practice, identification of memory leaks is more often attempted with techniques based on dynamic analysis [8, 48, 50, 68, 103, 107].

There are challenges and limitations in applying the tools and techniques developed based on dynamic analysis to JavaScript applications. First, most of these tools are tuned for a specific environment, such as Java/C++ programs, and cannot be used for JavaScript applications. Second, some of these approaches [39, 74, 107], require upfront information about the runtime behaviour of objects, which is not available in a dynamic typing language such as JavaScript. Third, many approaches based on dynamic analysis [29, 8] report allocation sites of the leaked objects; however, this information does not provide any insight on how to *fix* the leaks: a) because the allocation site and the location of the code that is causing the leak are in different parts of the application code [13], and b) existing approaches report a list of allocation sites, which can be confusing for the developer, because many allocation sites are related: for example, a leaky object may cause the allocation of many other objects; existing approaches report all those other allocation sites as leaky, in addition to the allocation site of the object that is the root cause of the leak.

With all the considerations in development mode, memory leaks may reach production and result in runtime errors and poor user experience. Looking at the current approaches for error detection and recovery during runtime, we realize that most current work focuses on the server-side [3, 10, 16, 26, 31, 46, 86]; however, we need new solutions on the client side, as the differences between the client and server environments introduce many new reliability challenges. First, in a client browser, there are several web applications running simultaneously, and errors in one web application may have some effects on the other applications. Second, approaches on the server side are meant to help system administrators who have more technical expertise than the ordinary user on the client side. Third, resources may be limited on the client side, which demands a low overhead for any technique taken for error detection and recovery at runtime. Finally, high interactivity on the client side of web applications makes transparent recovery challenging. In addition, it prioritizes user-perceived response time over throughput. Regarding the aforementioned differences between client and server environment, we need a new solution to make the client side of web applications dependable.

1.3 Solutions

This thesis consists of two parts. First, to address the lack of a general tool to detect memory leaks and guide developers to fix the leaks during the development phase of JavaScript applications, it introduces *LeakSpot*. Second, to improve memory efficiency of web applications and reduce the effects of memory leak errors at runtime, it develops a prototype tool, called *MemRed*. Figure 1.1 presents an overview of the solutions presented in this thesis.

1.3.1 Detection and Diagnosis of Memory Leaks

To find the causes of memory leaks during development of web applications and guide developers to fix the memory leak errors easily and quickly, we have developed *LeakSpot*. LeakSpot modifies the application code in a browser-agnostic way to make it possible to monitor all object allocations, all accesses made to an object, and all code locations where an object reference is created. LeakSpot reports the line numbers in the code that are allocating leaked objects. It also identifies the code locations where references are created to the objects but are not removed. To facilitate debugging and fixing, LeakSpot refines the list of leaky allocation sites by finding related allocation sites, reporting those that are the main cause of the leaks and removing those that are the side-effects of the allocation sites causing the leaks. In addition, for every leaked object, LeakSpot points out all the locations in the program where a reference is created to the object.

LeakSpot has several advantages over existing approaches for memory leak detection and diagnosis. First, LeakSpot provides more information for the developer than previous

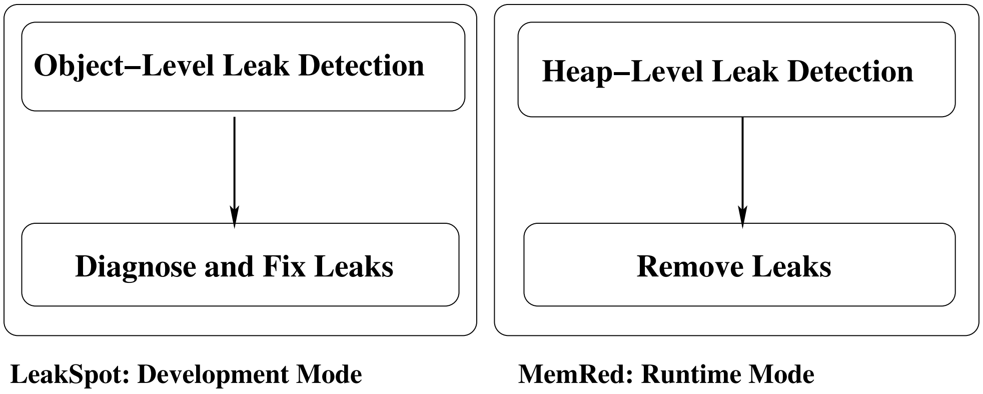

Object–Level Leak Detection
Diagnose and Fix Leaks
Heap–Level Leak Detection
Remove Leaks
LeakSpot: Development Mode
MemRed: Runtime Mode

Figure 1.1: Thesis Structure

approaches based on dynamic analysis [8, 50, 68, 107]. For every leaked object, LeakSpot not only reports the allocation site and last-use site, but also reports all code locations that create a reference to an object, which in turn keeps that object from being released. This information helps developers to find and fix the leaks by narrowing down the search space for finding the cause of the leaks. It shows not only where the leaked object is created but also *why* that object is being leaked. Second, it refines the list of allocation sites to distinguish real leaky allocation sites from those that are a side-effect of that leaky allocation site. Finally, monitoring all the points in the program where references are created allows finding the points in the program that accumulate objects without requiring upfront information about the object names and behaviour. As mentioned by Pienaar *et al.* [74], library data structures that hold references to objects are a common cause of memory leaks in JavaScript applications [74].

Compared to existing tools and solutions [39, 74] for JavaScript, LeakSpot provides more information and is not limited to specific libraries. First, the dynamic-analysis technique of LeakSpot makes it possible to know the type of objects during the leak-detection process, so unlike LeakFinder [39], it does not need the name of container data structures and does not rely on the annotations provided by the developer, as does JSWhiz [74]. In addition, it detects a larger class of leaked objects compared to the specific cases that are addressed in LeakFinder or JSWhiz. Specifically, closure-related memory leaks [19, 91], which are one of the causes of memory leaks in JavaScript applications, can be found using LeakSpot.

We have used LeakSpot to find and fix memory leaks in JavaScript benchmark and open-source web applications. In addition, we demonstrate the existence of memory leaks in large and popular web applications and identify the potential causes of said leaks. Moreover, we have measured the performance overhead of LeakSpot experimentally and discussed how the approach taken in LeakSpot, along with the characteristics of JavaScript applications, provides the opportunity to apply optimizations that make it possible to use LeakSpot beyond the development stage.

1.3.2 Runtime Management of Memory Leaks

To improve the memory efficiency of web applications and reduce the effects of leaked memory on application performance, we have developed *MemRed*. It is designed to work by monitoring the memory usage of web applications and then analyze the collected data to detect excessive memory usage that could indicate a memory leak. If the data analysis indicates the existence of an error, MemRed applies recovery actions at an appropriate time to hide the effects of memory leaks, such as poor response time and crashes, from the user.

The prototype of MemRed is implemented as an extension for the Chrome browser [42]. Although the Chrome browser provides high reliability by executing each web application within a separate process [77], there are several limitations to this approach. First, Chrome loads a page made of several iframes into the same process so that objects within different iframes are able to refer to each other; therefore, the iframes are not isolated [100]. Second, there is a limitation on the number of processes that Chrome will create, which depends on the available system resources. Upon reaching this limit, new pages share a process with currently opened pages. Third, an application within a separate process may suffer from errors or failures due to a bug in the application code, and process separation does not help in such scenarios. Therefore, we need new mechanisms for improving the reliability and availability of the client side of web applications. Our evaluation on MemRed shows the effectiveness of recovery actions in lowering the memory usage of web applications.

1.4 Contributions

In this thesis we make the following novel and significant contributions:

- We demonstrate the existence of memory leak in GMail application [82]. The memory leak in GMail is later reported by Google [56]. We demonstrate memory leaks in Facebook and Yahoo Mail, as well. To the best of our knowledge, we are the first one to demonstrate the specific memory leaks in these applications.

- We develop a profiler for JavaScript applications that makes it possible to collect data about object allocations, accesses, and references created on the objects during the runtime of applications.

- We develop a tool, LeakSpot, for detection and diagnosis of memory leaks in JavaScript applications. LeakSpot not only detects leaked objects, but also identifies the areas of code for a developer to examine for removing the causes of the leaks. More specifically, similar to previous work, LeakSpot reports leaky allocation sites and last-use site of the leaked objects. In addition, LeakSpot reports: (a) the allocation-site graph, which reduces the number of allocation sites to just those that are the likely cause of the leak and/or are useful in finding the cause of the leak, (b) all the accumulation sites which are the points in the program where objects are kept alive by unwanted references, e.g., from a library data structure (c) all the locations in the code that a reference is created to the objects which is useful for finding the unwanted references that prevent the leaked objects from being garbage collected. This information guides the developer to fix the leaks quickly and easily.

- We evaluate LeakSpot on complex JavaScript benchmarks and open-source web applications, thereby demonstrating the effectiveness of LeakSpot in finding the leaks and guiding the developer in fixing the memory leaks quickly and easily.

- We evaluate LeakSpot on large and popular web applications and point out the potential locations in the code that are problematic.

- We develop MemRed [81, 82], a prototype tool for online detection of memory leaks and recovery of the system from the effects of excessive memory use during web application runtimes. The results of an empirical study on a real-world web application as well as a benchmark application demonstrate the effectiveness of MemRed in improving the memory use of web applications.

1.5 Impact and Potential Impact

Using LeakSpot, we were able to find and easily fix two instances of memory leaks in Octane benchmark [70], one of the most complex JavaScript benchmarks. The fix for one of the memory leaks is already confirmed by developers of Octane and submitting the fix for the second one is in progress.

Also, using LeakSpot we were able to find and fix two instances of memory leaks in the TodoMVC [93] project, a set of open-source web applications that are created to help developers in choosing the appropriate JavaScript library. We submitted patches for these memory leaks and these fixes are already merged into the main repository of the project. This experiment allowed us to test LeakSpot on many different JavaScript libraries.

In addition, LeakSpot is used to find memory leaks in large web applications such as GMail. While this experiment demonstrated the scalability of LeakSpot, we could not fix the leak since the obtained code is obfuscated and we do not have access to the unobfuscated code; however, the fact that LeakSpot was able to guide us in fixing the memory leaks in Octane benchmark and open-source web applications assures us that it would be useful for other web applications. In the case of GMail the number of leaky allocation sites is reduced to 6 which strongly suggests that it would be quick to check the relevant sites and fix the leaks.

As the potential impacts of this work, LeakSpot facilitates developing leak-free applications by helping the developers of JavaScript applications find the causes of memory leaks. Preventing leaks results in more memory efficient web applications, which, in turn, results in applications with higher performance. In addition, the technique proposed for managing leaks at runtime have the potential to improve user experience.

1.6 Overview of the Thesis

The rest of this thesis is organized as follows. It starts by discussing the background and related work in Chapter 2. Then, it presents the implementation and evaluation of LeakSpot, including the details of the profiler, in Chapter 3. Next, it explains the approach that is taken for runtime management of memory leaks in Chapter 4. Finally, it concludes in Chapter 5 by summarizing the thesis and describing future work.

Chapter 2

Background and Related Work

This Chapter first provides the definitions and background information needed for full understanding of the thesis, including full description of the system, fault, and failure models. Next, previous approaches for solving memory leaks are presented with the explanation of their differences from and similarities to the work in this thesis.

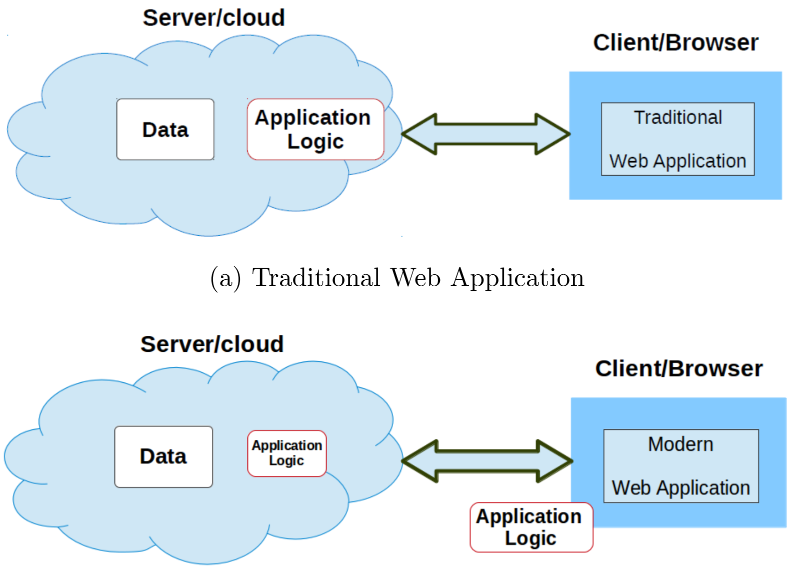
2.1 System Model

The system model specifies the system under study. This section describes the system model in this work, including its components and the motivations for choosing the specific model.

This thesis focuses on the client side of *modern web applications*[1]. In recent years, the architecture of web applications has changed tremendously, with the migration of large parts of business logic from the server side to the client side [94]. As shown in Figure 2.1, in modern web applications, large parts of the code are moved to the client side, where the code is written in JavaScript and run in the browser. The main technologies on the client side of modern web applications are:

- HTML and CSS: These technologies are used for information presentation and user-interface development. Newer versions of HTML, such as HTML5 [99], provide the opportunity to have rich user interface features in web applications.

[1] Also called AJAX (Asynchronous JavaScript and XML) applications, Rich Internet Applications (RIA), or single-page web applications.



(a) Traditional Web Application

(b) Modern Web Application

Figure 2.1: Architecture of traditional and modern web applications

- Document Object Model (DOM): DOM is a structural representation of the HTML. It is structured as a tree, whose nodes are objects with properties and methods. The client side of JavaScript applications is composed of JavaScript code embedded into HTML pages. To interact with the pages and access and modify HTML documents dynamically in JavaScript, DOM APIs are used. In typical browsers, the JavaScript version of the DOM API is provided via the `document` host object. The methods and properties supported in DOMs are defined in DOM specifications, i.e., documents developed by W3C group[2] to standardize access and manipulation of HTML and XML objects.

- Asynchronous JavaScript and XML (AJAX): This technology is used to exchange data between the client and server in an asynchronous way, while XML is used to wrap data.

- Javascript: This is the main language used for browser development. Message handlers or user-event handlers, which change the content or structure of a web page, are implemented using JavaScript.

[2] The main international standard organization for the World Wide Web (WWW).

The changes in the architectures and development technologies of web applications have made modern web applications more responsive than traditional web applications, so they are more like desktop applications: First, in modern web applications, computation has moved closer to the client, avoiding unnecessary network round-trips for frequent user actions. Second, asynchronous communication between client and server as well as modifications to a page using DOM APIs allow parts of the page to be updated, instead of refreshing the whole page upon a change in part of the page. Finally, the existence of a rich user interface in modern web applications makes users feel more like they are working with a desktop application. Examples of modern web application are GMail [37] and Google Docs [38], in which lots of application code is run on the client browser. GMail allows the user to handle email through a browser and Google Docs allows the user to write documents or use a spreadsheet through a browser.

Ideally, AJAX applications are independent of a specific operating system, virtual machine, plug-ins in the browser, or external programs installed in the user's desktop computer. These applications need only a modern browser with access to the Internet on the client side; however, some aspects of the HTML and JavaScript environments are not implemented consistently among browsers, so these applications are not completely independent of the browser in which they run [51]. To hide the cross-browser differences, JavaScript libraries and frameworks [18, 36, 45, 49] are used for application development, which can introduce some problems in the application, including memory leaks, as will be discussed in Section 2.3.2.

2.1.1 Motivation

This section presents the motivation for working on the client side of web applications. The changes in architecture and development technologies of web applications have made the client side more complex compared to traditional web applications. Recent studies of modern AJAX-based web applications indicate that front-end execution contributes 88%-95% of execution time, depending on whether the browser cache is full or empty [53]. As the client side has grown, so too has the need for mechanisms to improve the *dependability* of the applications by improving the reliability and availability of the client side. In addition, as users tend to keep their browsers open with many applications for a long time, an error in one web application can affect the reliability and availability of the whole browser [77] and the underlying system.

2.2 Threats to System Dependability

Threats to system dependability are faults, errors, and failures [7]. A fault is a flaw in the system that results in an error when activated. An error is the corruption of the system state and a failure is a deviation in system behavior that can be observed by the user. The system may misbehave but there is no failure as long as it does not generate incorrect output. Figure 2.2 shows the relationship between faults, errors, and failures. Symptoms are the side effects of errors in the system; e.g., the existence of memory leak in one application (fault) can cause memory usage of the system to go up (sysmptom). The following subsection presents the assumptions made about the faults and failures explored in this thesis.

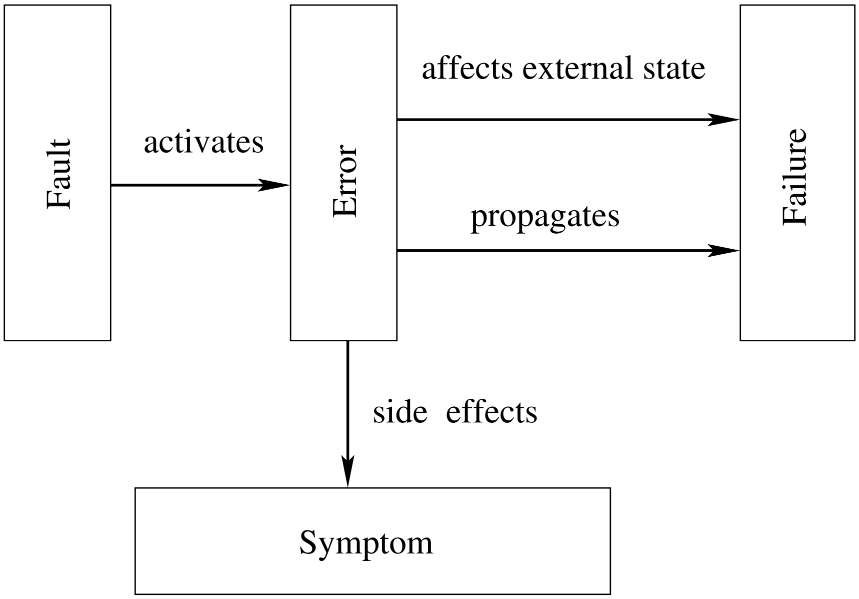


Figure 2.2: Relationship between fault, error, and failure [83]

2.3 Fault Model

A *fault model* is the set of assumptions that a designer makes about the faults in a system. Faults in software systems can be classified into *Bohrbugs* and *Heisenbugs* based on their phase of creation [95]. Bohrbugs are permanent design faults that can be identified easily and removed during the testing and debugging phase of the software life cycle. Bohrbugs are also referred to as deterministic faults since they result in repeatable failure. Heisenbugs are design faults which are activated rarely or are not easily reproducible. They are extremely dependent on the operating environment, such as other programs, operating system, and hardware resources. The occurrence of failures due to Heisenbugs cannot be predicted. If a fault is activated in one run of the system, it may not be activated after a system restart. Therefore, these faults result in transient failures.

Faults that result in software aging are labeled as *aging-related faults* [95]. Aging-related faults can fall under Bohrbugs or Heisenbugs depending on whether the failure is deterministic (repeatable) or transient. Software-aging faults degrade the system gradually and may eventually result in crashes or poor response time [24]. Memory leaks, memory fragmentation, storage-space fragmentation, and data corruption are some examples of software-aging issues. Like any other software system, web applications are prone to different types of faults. This thesis focuses on memory leak faults. In the next section, the definition of a memory leak used in this thesis will be specified in more detail.

2.3.1 Memory Leak

Memory leaks, that is gradual loss of memory, are a threat to the reliability of software systems. They result in performance degradation and may cause the program to crash, in the worst case. Memory leaks are mainly occurred because of the two following reasons [50, 13]:

- Lost pointers: losing all pointers to objects that the program forgets to free.
- Unnecessary references: keeping pointers to objects the program never uses again.

In languages with explicit memory management, such as C/C++, objects may be leaked in both ways mentioned above; while, in managed languages, such as Java/JavaScript, objects are leaked because they are kept alive by unnecessary references. In managed languages, memory management is done automatically using garbage collection; therefor, such memory leaks caused by lost pointers do not happen. The two most common garbage collection techniques are:

- Reference-counting garbage collection: This algorithm considers an object as garbage if no reference is pointing to it. An object is said to reference another object if the former has an access to the latter, either implicitly or explicitly. For instance, a

```
var div = document.createElement("div");
div.onclick = function(){
  doSomething();
 };
```

Figure 2.3: Example of a memory leak due to circular references

JavaScript object may have a reference to its prototype (an implicit reference) and to its properties values (an explicit reference).

This algorithm has the limitation that if objects reference one another and form a cycle, they may not be used anymore and yet not considered a garbage. For example, Internet Explorer 6, 7 are known to have a reference-counting garbage collector for DOM objects. For both, a common pattern is known to generate memory leaks systematically. Figure 2.3 demonstrates this pattern of memory leak in which a cycle is created between a DOM element and a JavaScript function. The `div` variable (DOM object) has a reference to the event handler (JavaScript object) via its `onclick` property. The handler also has a reference to `div` since the `div` variable can be accessed within the function scope. The cycle created between `div` and the handler function will cause both objects not to be garbage-collected, which results in a memory leak. This pattern of memory leaks caused by the interaction between DOM and JavaScript has already been solved by major browsers.

- Mark-and-sweep garbage collection: This algorithm assumes the knowledge of a set of objects called roots, e.g., global variables or stack variables. The garbage collector periodically starts from these roots and finds all the objects that are reachable from the roots. The objects that are not reachable from the garbage collection roots are garbage and are collected by the garbage collector. This algorithm works better than the the reference-counting garbage collector since it removes cycles. For this approach to work, the objects need to be made explicitly unreachable. If the programmer forgets to remove some references, then unneeded memory remains alive on the heap, resulting in a memory leak.

Thus, a memory leak in a garbage-collected language occurs when a program maintains references to objects that it no longer needs, preventing the garbage collector from reclaiming space. Having mentioned the different garbage collection techniques, the garbage is next categorized into two types:

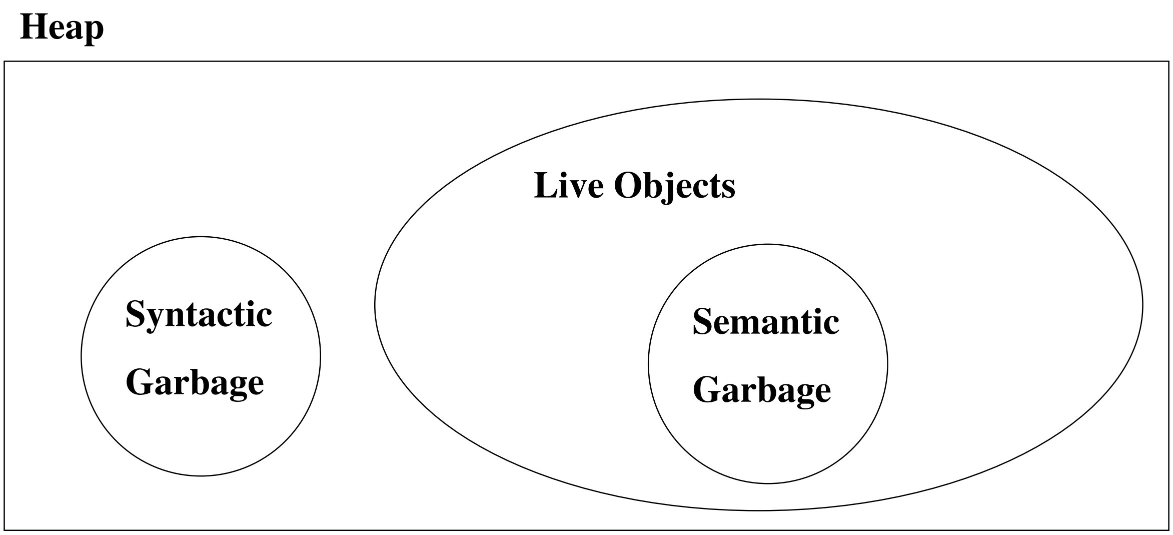


Figure 2.4: Live objects and garbage on the heap

- Semantic garbage: Any object or data which will never be accessed by a running program, for any combination of inputs to the program.
- Syntactic garbage: Objects or data within a program's memory space that are unreachable from the program's root sets.

Garbage collectors collect syntactic garbage but in this work we are interested in finding semantic garbage. The semantic garbage is maintained on the heap by unwanted references. Objects and/or data which are not syntactic garbage are said to be live. Figure 2.4 shows the relationship between different types of garbage and live objects on the heap.

Detecting unused memory is undecidable The problem of precisely identifying unused memory is an undecidable problem [96, 105]. Consider a simple program as follows:

```
     allocate an object X
     run an arbitrary program P
     use X
```

The above program uses X if and only if P finishes. Determining whether P finishes or not would require the halting problem[3] to be decidable. Since, we know the halting problem is

[3] In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run

undecidable, we conclude that the problem of identifying unused memory is undecidable.

2.3.2 Examples of Memory Leaks

This section provides examples of memory leaks in JavaScript. The unnecessary references that prevent an object from being released, and thus result in its being leaked, are maintained because of programmer errors. Some characteristics of the language and development environment may lead the developers to easily forget to remove references. Two of the most common reasons for memory leaks in JavaScript are the data structures in the JavaScript libraries and the closure variables that are widely used in JavaScript.

Library data structures As mentioned by Pienaar *et al.* [74], JavaScript libraries and development environments [18, 36, 45, 49], which are mainly used to hide cross-browser differences and to ease the development of large web applications, are one of the main causes of memory leaks [74]. Figure 2.5 shows an example where leaked objects are retained unintentionally by a container (a data structure in the library code) [39]. This is the example code that the developers of LeakFinder [39] used to demonstrate how their tool works. In this code snippet, objects `handle2` and `handle3` are leaks since they are intended to be deleted by the programmer (set to null in lines 22 and 23), but are retained live by the library data structure (`goog.Disposable.instances_`). These objects are an instance of MyObj, which inherits from goog.Disposable. In its constructor, good.Disposable assigns the objects to a container (lines 3 and 4).

JavaScript closures Another common cause of memory leaks in JavaScript involves closures [25, 72]. JavaScript is a functional programming language and its functions are closures, i.e., function objects get access to variables defined in their enclosing scope, even when that scope is finished. Local variables captured by a closure are garbage collected once the function they are defined in has finished and **all** functions defined inside their scope are themselves garbage collected. Figure 2.6 shows a simplified example. The `closure` function, which is returned by `ClosureGenerator`, encloses `closurevariable`. Therefore the memory allocated by `closurevariable` is not released as long as `reference` is alive, even though `closurevariable` is not used by the `closure` function. Please note that although the `closurevariable` is not used by the `closure` function, it is used in the other function, `leakgenerator`. Since `leakgenerator` and `closure` share

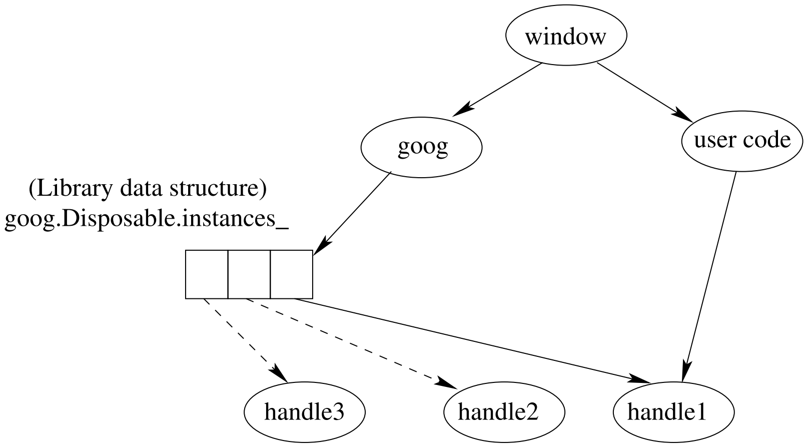
forever.

```
------Library Code---------
goog.Disposable = function() {
    goog.Disposable.instances_[goog.getUid(this)]
        = this;
};
------User Code------------
MyObj = function() {
  goog.base(this);
}
goog.inherits(MyObj, goog.Disposable);
MyObjCreator = function() {}
MyObjCreator.prototype.Create = function() {
    return new MyObj();
}
var creator = new MyObjCreator();

// Not a leak
var handle = creator.Create();

// Objects habdle2 and handle3 are leak
var handle2 = creator.Create();
var handle3 = creator.Create();
handle2 = null;
handle3 = null;
```

(a) Simplified code of memory leak



(b) Heap view

Figure 2.5: Example of the memory leak used in LeakFinder

```
function ClosureGenerator() {
    var closurevariable;
    function closure(){}
    return closure;

    function leakgenerator() {
        var temp = closurevariable;
    }
    leakgenerator();
}
var reference = ClosureGenerator();
```

Figure 2.6: Example of a memory leak caused by closures in JavaScript

lexical scope, `closurevariable` is enclosed by both `closure` and `leakgenerator`. if `closurevariable` were entirely unused, recent JavaScript engines would optimize it out of existence. Since the objects that are bound to the closure may not be used at all, a staleness-based heuristic is a good approach for finding memory leaks caused by closures.

2.4 Failure Model

The failure model specifies the assumptions about the system behaviour in case of failure [89]. The circumstances of when and how a fault causes a failure and the behaviour of the failure determine what a system must do to tolerate the failure.

The work in this thesis focuses on crash failures and timing failures [6, 15]. A crash failure happens when the system stops behaving correctly, e.g., a web application fails to produce any output. A timing failure occurs when the system responds correctly but in an unacceptable time; e.g., a web application fails to respond to a user request in an acceptable time. Performance failures are regarded as timing failures. Memory leaks caused by unnecessary references degrade the performance of an application by increasing memory requirements and consequently garbage collector workload [50]. In addition, a growing data structure with unused parts may cause the program to run out of memory and crash. Even if a growing data structure is not a true leak, application reliability and performance may suffer.

2.5 Related Work

The work in this thesis builds on a large body of work on memory leak detection and diagnosis in other programming languages. It also draws inspiration from work on error detection and recovery on the server side of applications. This section presents the preceding work that is related to and that inspired the work in this thesis, highlighting the similarities and differences.

2.5.1 Memory Bloat

Memory bloat is a more general problem than memory leaks and specifies the inefficient use of memory in software systems. The work targeting memory bloat attempts to find, remove, and prevent performance problems due to inefficient use of memory in the application [106]. Dufour *et al.* [21] assume temporary data structures as a source of inefficient memory usage and propose a tool for identifying excessive use of temporary data structures. Mitchell *et al.* [61] propose an approach for distinguishing effective from excessive use of memory by making a health signature based on data from dozens of benchmarks and applications. Unused objects remaining alive due to unwanted references are a special kind of memory bloat that can be detected by LeakSpot.

2.5.2 Memory Leak Detection in Other Programming Languages

The memory leak problem has been addressed extensively in the context of other languages such as Java/C++. Because of the limitations of static analysis, heuristics based on dynamic analysis are used. The different dynamic-analysis approaches use various techniques such as heap growth [50, 60], heap analysis [39], object staleness [8, 29], or ownership profiling [76] to determine leaked objects.

Leakbot [60] and Cork [50] are based on heap growth. Cork uses types whose number of instances continue to grow to determine the leaky types in the application. It then needs extra work by the developer to find the cause of the leak since knowing the types of objects is not sufficient for finding the causes of the leaks. Leakbot works by analyzing the heap to find a suspicious data structure and then tracks the specific region in the data structure to find any growth. Since LeakSpot collects runtime information about objects, it has detailed information that can be useful in locating the leaks. In addition, LeakSpot could be slightly modified to find problematic data structures automatically by monitoring patterns of reference creation Moreover, the approaches based on heap growth must be implemented at the JavaScript engine level, which makes the portability of these approaches difficult.

Xu *et al.* [107] assume that specific data structures are causing the leaks and apply heuristics to find those data structures. They monitor any addition and deletion to the data structures used in the application by annotating the corresponding library functions. The data structures for which objects are added but are not deleted are suspected of being leaky. This approach cannot be applied in JavaScript since there is no prior knowledge about the runtime behaviour of an object because of the dynamic typing in JavaScript. Although LeakSpot does not find problematic data structures, it finds the problematic points in the program, e.g., the points that objects are added to a problematic data structure.

Bell [8] and SWAT [29] report stale objects as leaked objects. They monitor object allocations and accesses by instrumenting the Java virtual machine and report those objects that have not been used for a long time as stale. LeakSpot borrows the idea of staleness-based heuristics from Bell and SWAT, and makes it possible to use these techniques in the context of JavaScript, but LeakSpot makes different design decisions. First, Bell [8] and SWAT [29] encode allocation sites and last use sites in the object header, which limits the amount of information that can be collected about an object; however, the code-level instrumentation of LeakSpot makes it possible to record all the modifications and accesses to objects. This detailed information guides developers in fixing the memory leaks quickly and easily. Second, to reduce instrumentation overhead SWAT [29] uses adaptive profiling. As a consequence, this approach does not record all accesses to an object, which can result in false positive results; however, the different design decision of LeakSpot make it possible to apply optimizations that reduce performance overhead without decreasing leak-detection accuracy. Applying the optimizations makes it possible to use LeakSpot in an online setting. In long-running applications, small leaks can take a long time to manifest. Such bugs are notoriously difficult to find, so having an online tool would be helpful in such cases.

The object ownership profiling [76] technique has been used to find memory leaks in Java applications. This technique uses different information about an object such as its size, allocation interval, active interval, method calls, and field accesses, as well as the heap structure to find leaked objects. While this approach can accurately locate leaked objects, the different categories of information used by the technique makes it more heavyweight than LeakSpot.

Leakpoint [13] finds memory leaks in applications developed using C++ and proposes fixes to the developers. To find memory leaks caused by lost pointers, Leakpoint tracks the number of references created to objects or removed from the objects to determine the leaked objects. To find memory leaks caused by unwanted references, Leakpoint reports all the objects that are not deallocated at the end of program execution. Monitoring reference creation, removal, and object use make it possible for Leakpoint to propose fixes to the developer. Similarly, LeakSpot guides the developer in fixing the leaks by monitoring reference creations and modifications as well as accesses to the objects.

A series of tools have been developed [62, 63] to help in finding memory leaks in the Firefox browser, an application developed using C++. For example, LeakGauge [62] is a tool that can be used to detect certain kinds of leaks in Gecko [62]. It works by instrumenting Gecko (which is developed in C++) and processing the generated log file to find leaked DOM objects; however, the leaks found by LeakGauge are the leaks that are caused by the browser, an application developed in C++. These are not JavaScript memory leaks.

2.5.3 Memory Leak Detection in JavaScript

In the context of JavaScript, JSWhiz [48], which is an approach based on static analysis, has been proposed. JSWhiz is based on the assumption that objects are allocated and deallocated manually. This assumption does not hold in general for JavaScript, which manages memory automatically using garbage collection. JSWhiz addresses programming environments where object allocations and deletions are done explicitly using library functions such as the Closure library [36]. In these cases, objects should be explicitly removed, and a failure to do so results in a memory leak. JSWhiz relies on annotations provided by the programmer to infer object types. LeakSpot works more generally based on standard methods of object allocation, access, and reference creation in JavaScript. Unlike JSWhiz, LeakSpot does not require any specific action during code development, i.e., a code annotated with types information. In addition, it does not rely on specific mechanisms for object allocation and deallocation.

LeakFinder [39], which is a tool for finding memory leaks in JavaScript applications, works by analyzing the heap of applications. It is based on the assumption that leaked objects are retained *only* by library data structures. An object is said to be leaked if all the paths from roots to the object go through the library data structures. Having the heap graph of the application as well as the name of those library data structures, it traverses the heap graph of the application to find leaked objects, i.e., objects that that are retained *only* from the specific data structures. This approach assumes knowledge of the name of the suspected data structure in the library whereas LeakSpot can automatically find suspected points in the program that objects are added to the data structures. In addition, LeakFinder does not report the location of the leaked objects in the code.

Tools like Chrome Developer Tools [34] provide a good view of the objects on the heap, but they cannot be used for finding the causes of the leaks. Approaches like three-way heap snapshots [74] can be used for finding the leaked objects under a specific scenario, without pointing out the causes of the leaks; however, LeakSpot can automatically find the leaked objects and point to the locations in the code that are causing the leaks.

There are a large number of tools for profiling Node.js applications [28]. These tools provide the same profiling data for Node.js [43] applications that Chrome Developer Tools provide for browser applications. While these tools provide a good view of the heap, they have the same limitations as Chrome developer tools. They need intensive work by the developers to find the cause of the leaks.

2.5.4 Web Application Profiling

To monitor web applications, Richards *et al.* [79] instrument the Javascript engine of Safari and collect data consisting of read and write activities on objects. These data are not enough for detecting memory leaks in web applications since they do not contain lifetime of the objects. In addition, this technique is limited to a specific browser and JavaScript engine.

Kiciman *et al.* [51] have developed a framework for profiling web applications through a proxy called AjaxScope. This tool is not available online. AjaxScope [51] instruments Javascript code before sending it to the client. The instrumented code running on the client browser sends data about application states and user interactions to the proxy, which analyzes the data and sends the report back to the server upon detecting an error. The authors have discussed the potential of their tool, AjaxScope, for memory leak detection; however, they only discussed a specific kind of leak that happens because of circular references, a situation that has been resolved in many browsers. In this work, in addition to developing the proxy-based profiler, we provide modifications that allow us to collect a rich set of runtime data that is useful for detecting and diagnosing memory leaks.

JavaScript instrumentation has been done for different purposes such as program analysis [85] or record and replay of web applications [78], and has inspired the specific instrumentation for memory leak detection in LeakSpot.

2.5.5 Runtime Error Detection and Recovery

The goal of runtime management of errors is to make the web applications dependable by making them capable of detecting errors (predicting failure) and then taking action to return the system to a healthy state automatically. In particular, we need tools for detecting excessive memory usage and taking action to remove its effects.

To predict failures, prior work on the server side has learned and modelled system behaviour based on system metric data or failure data. The models are then used during runtime to detect errors or predict failures. Grottke *et al.* [26] modelled system behaviour while the server is under artificial workload and then used this model at runtime to predict system failures. Alonso *et al.* [3] collected failure data and then used machine-learning techniques to estimate time-to-failure of a web server. These approaches cannot be used for failure prediction on the client side of web applications, where many different applications are running inside the browser and modelling the behaviour of one specific application is not useful.

Web applications are subjected to different types of failures. Functional errors in web applications have been studied by Ocariza *et al.* [69]. They categorize errors in Javascript-based web applications and study the correlation between application properties and failure frequencies. Failures due to security bugs have been studied in the past [27]. To diagnose errors, Mugshot [58] and WaRR [4] record events on a Javascript page and replay them after failure to find the causes of the failures. In this thesis, we focus on excessive memory usage due to memory leaks that degrade software systems gradually and result in poor response times or crashes.

To return the system to a healthy state, we need to apply recovery actions. The strategy for recovery adopted in this work is based on recovery-oriented computing [73], which tries to minimize the mean-time-to-recovery. This approach is in contrast to traditional fault-tolerant systems that focus on minimizing the mean-time-to-failure by redesigning the system. More specifically, instead of changing the browser or web applications to make the client side of web applications fault-tolerant, our goal is to make the web applications capable of recovering from failures automatically. Software rejuvenation [10, 11, 14, 31], which involves rebooting the whole or part of a system periodically to clean its state, has been used before. It inspired us to use similar recovery actions for web applications. The important point is that we need finer-grain actions on the client side. For example, machine reboot is a reasonable recovery action on the server [46] because it will have little effect on system throughput if it can take advantage of replication; however, machine reboot is not an appropriate recovery action on the client since we have only one machine. In Microreboot [10], the authors proposed to reboot a part of the system; however, to achieve this goal, they had to change the application server architecture, which limits their approach to custom applications. In the browser environment, we can leverage the browser structure to implement recovery actions that reboot a specific part of the system.

Chapter 3

Detection and Diagnosis of Memory Leaks in JavaScript Applications

This chapter presents the design, implementation, and evaluation of LeakSpot, a tool developed to address memory leaks during development of JavaScript applications. It starts by explaining a running example that will be used throughout this chapter to describe different aspects of LeakSpot. This example is a memory leak in an open-source web application that was found and fixed using LeakSpot.

3.1 Running Example

This section describes a memory leak example that was found and fixed in the Todo-Dojo [93] application[1]. This application was developed using the Model-View-Controller (MVC) concept in the the Dojo library [18]. Todo-Dojo is part of the TodoMVC [93] project, in which the same application is developed using the MVC concept with different JavaScript libraries. The leak discussed here happens when we repeatedly add Todo tasks to the list and then remove them.

Figures 3.1 and 3.2 show simplified versions of the code required to understand the memory leak in the Todo-Dojo application, including the relevant parts of the Dojo library in Figure 3.1 and the developer code in Figure 3.2. To make it easier to refer to lines of code in the library and developer code, the line numbers in the two figures do not overlap. Therefore, when referring to a part of the code, only a line number is used.

[1] We sent the fixes to the developer and they changed the repository accordingly.

```
// dijit.registry module
define([array, window], function (array, win) {
  var hash = {};
  var registry = {
  add: function(widget){
    hash[widget.id] = widget;
    this.length++;
  },
  remove: function(id) {
    delete hash[id];
    this.length--;
  }
}
//_WidgetBase Module
define([], function() {
  var _WidgetBase = declare("dijit._WidgetBase", [], {
    id: "",
    create: function() {
      registry.add(this);
    },
    destroy: function() {
      registry.remove(this.id);
    }
  });
  return _WidgetBase
});
```

Figure 3.1: Simplified code of the memory leak in Todo-Dojo application: library code

```
//app.html
<ul id="todo-list" data-dojo-type="todo/TodoList"
data-dojo-props="children: at(${id}_listCtrl, "model")"
data-mvc-child-props="uniqueId: at(this.target, "uniqueId")"
</ul>

//CssToggleWidget Module
define([_WidgetBase], function( _WidgetBase) {
  return declare(_WidgetBase, {
    ...
  });
});
//TodoList Module
define([CssToggleWidget, WidgetList], function (WidgetList, CssToggleWidget) {
  return declare([WidgetList,...], {
    childClz: declare([CssToggleWidget,..], {
      onRemoveClick: function(){
        this.parent.listCtrl.removeItem(this.uniqueId);
        this.destroy(); //To fix the leak
      }
    });
  });
});
//TodoListRefController Module
define([ModelRefController], function( ModelRefController) {
  return declare(ModelRefController, {
    addItem: function(title) {
      this[this._refModelProp].push(new Stateful({title: title, completed: 'false'}));
    },
    removeItem: function(uniqueId) {
      var model = this[this._refModelProp];
      var index = find_the_index_to_be_removed(uniqueId);
      this[this._refModelProp].splice(index, 1);
    });
  });
```

Figure 3.2: Simplified code of the memory leak in Todo-Dojo application: developer code

Whenever a task is added to a list, one item is added to the data model (line 67). Also, an object of type TodoList (line 53 defines TodoList) is created and added to the list of widgets (user interface components representing the task) on the page (lines 41-43). The TodoList module inherits the CssToggleWidget module which inherits the _WidgetBase module. Therefore, whenever a task is added to the list, an object of type _WidgetBase is created and a reference to the object is created from the hash data structure in the dijit.registry module (line 15 creates the reference). Whenever a task is removed from the list, the item is removed from the model list (line 72), while the corresponding widget from the registry is *not* removed, causing the leak. To fix this leak, the developer needs to remove the TodoList widget by calling the destroy function (this is added at line 58 of the code). When called, the destroy function calls the remove function (line 18) which removes the reference from the hash data structure (lines 19 and 20).

For this example leak, LeakSpot returned 10 leaky allocation sites, one of them the allocation site in line 67 and the others at various locations in the library code. In addition, LeakSpot reported 11 reference sites, one of them is the reference created at line 15. Having this information helped us to fix this leak quickly and easily. We did not have any prior knowledge about the Todo-Dojo application.

Figure 3.3 shows a simplified view of the memory leak on the heap of the application. Whenever a task is added, two objects are created: T* and M*, e.g., T1 and M1, and two

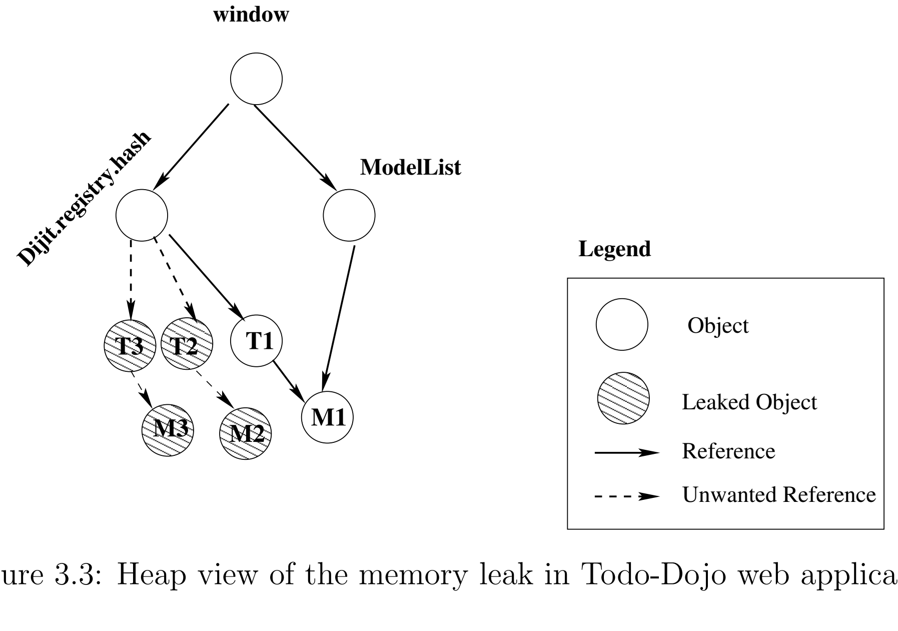


Figure 3.3: Heap view of the memory leak in Todo-Dojo web application

references are created for each object: one from ModelList (line 67 in Figure 3.2) and one from the hash data structure (line 15 in Figure 3.1); however, whenever a task is removed, only the reference from ModelList to M* is removed (line 57 in Figure 3.2). The reference from dijit.registry.hash to T* is not removed, thereby preventing the objects from being released, e.g T2, M2, T3, and M3, which are leaked. This example demonstrates the type of leak that garbage collection cannot handle, since garbage collection collects only unreachable objects, whereas the unused objects here (T2, M2, T3, and M3) are reachable from garbage collection roots (window). Since determining the unused objects is an undecidable problem [76], all the solutions resort to heuristics.

This example is a good representative of the type of memory leaks that are common in JavaScript applications, i.e., memory leaks caused by unwanted references. It is similar to the synthetic example that the authors of LeakFinder [39] used to demonstrate the effectiveness of their tool, as mentioned in Chapter 2; however, to find such a leak, LeakFinder needs the name of the library data structure (e.g., dijit.registry.hash). In addition, LeakFinder does not point to the locations in the code that allocate the leaked objects or create references to the leaked objects. It only identifies leaked objects on the heap, i.e., it reports objects T2, T3, M2, and M3 as leaks. It requires extra effort by the developer to find the relevant location in the code. JSWhiz does not work for this example, as it works only for the code developed using the Closure library [36]. In addition, this example demonstrates that LeakSpot does not rely on any specific mechanism for object allocation. The leaked objects in the example are allocated using a mechanism specific to the Dojo library. The TodoList objects are allocated by setting specific properties of an HTML element, e.g., data-dojo-type and data-dojo-props in lines 41 to 43 in Figure 3.2, which is a mechanism specific to the Dojo library. This example demonstrates that LeakSpot does not rely on any specific mechanism of object allocation, unlike JSWhiz, and can work in general.

3.2 System Overview

LeakSpot is composed of three main components: *Profiler*, *Logger*, and *Leak Reporter*, as shown in Figure 3.4. Profiler modifies the web application code to make it possible to monitor all allocations, accesses made to the objects, and references created on objects. The modified code, which is generated in the proxy, is sent to the client browser, along with a profiler library that contains the definitions and implementations of logging functions. The Logger records all the profiling data at runtime and sends the records back to the proxy periodically. Leak Reporter analyzes the profiling data to find those parts of the

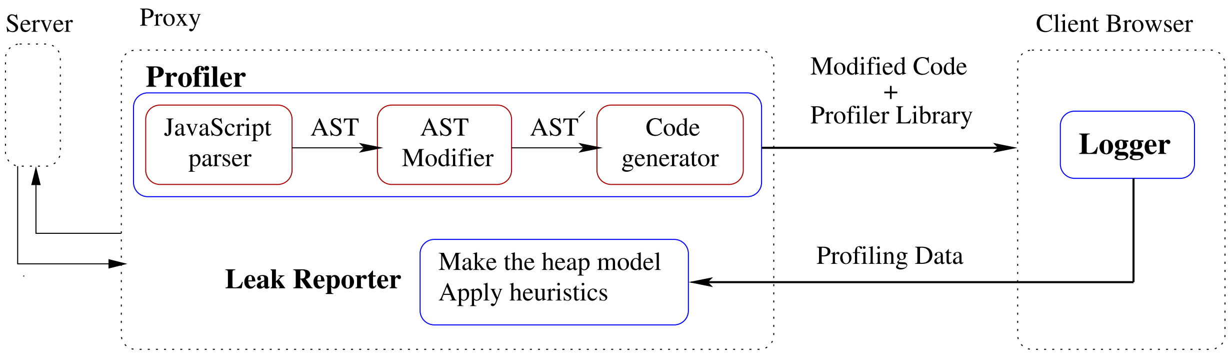


Figure 3.4: LeakSpot architecture

program that are generating leaked objects. The proxy is meant to sit anywhere between the client and the server. In cases in which the the developer has access to the server side, placing the proxy on the server side reduces performance overhead, as discussed in section 3.6.5. In the following, the components of LeakSpot are explained in more detail.

3.2.1 Profiler

Profiler modifies the code by intercepting object allocations, accesses, and reference creations in the code. Instrumenting the code provides a mapping between objects on the heap and the location in the code and the time during execution they are allocated, accessed, modified, or referenced. As shown in Figure 3.4, to instrument the JavaScript code, the profiler parses the code to generate the Abstract Syntax Tree (AST). It then traverses the AST and inserts required changes. Finally, it generates the new code from the modified AST.

Dynamically Generated Code In JavaScript, code may be generated during runtime, e.g., in the `eval` function. The dynamically generated code is not modified by the proxy in the first pass; therefore, it is sent to the proxy to be modified during runtime. The `eval` function is wrapped, so that it sends the code to the proxy and waits for the modified code, before executing it. The functionality needed to wrap the `eval` function is included in the profiler library.

3.3 Logger

To make the modified code run flawlessly in the browser, the proxy injects the profiler library into every HTML page/iframe. The profiler library provides several logging functions. A logging function takes as its parameter the object, its properties (if applicable), and the line number of the code where the action was performed on the object. When executed, a logging function assigns a unique id to the object, if an id has not been assigned before. The id is assigned to the object by dynamically adding a property named `__objectID__` to the object. The id is added as a non-enumerable property so that it is not visible to the code during execution, i.e., when iterating through the properties of an object, the id is not seen as a property. The assigned object id will be used in the Leak Reporter to track all activities on the object.

Following its execution, the logger function inserts the profiling data into a memory buffer. Each record of the profiling data is composed of three main pieces of information: first, the location id (LN), which specifies the line number in the code where the action has taken place; second, the type of action on the object, i.e., whether an object is allocated, accessed, modified, or has a reference created on it; finally, the id that is assigned to the object by the Logger. The Logger periodically sends the profiling data to the proxy.

3.4 Leak Reporter

To determine leaked objects, LeakSpot first makes a heap model based on the profiling data. Then it applies heuristics to find leaked objects and subsequently, the locations of the code that are allocating the leaked objects or preventing the objects from being released. A more-detailed description of how LeakSpot determines leaked objects follows.

3.4.1 Heap Model

To keep track of the object allocations, accesses, and references created on an object, LeakSpot creates a heap model based on the profiling data. Data structures used in the proxy to keep the heap model are shown in Figure 3.5. The heap model implements three interfaces: ObjectAllocated, ObjectAccessed, and ReferenceCreated. Upon receiving a log record indicating that an object is allocated, the function implementing the ObjectAllocated interface is called, which creates an object descriptor and saves the information about the object into a map data structure, ObjectRecencyMap, as shown in Figure 3.5. This

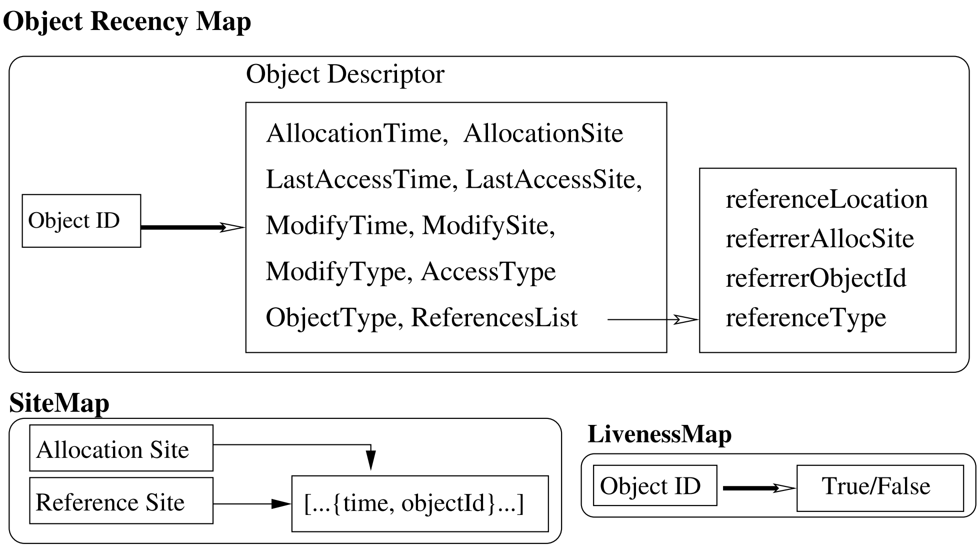


Figure 3.5: Data structures in Leak Reporter

data structure maps an objectID to the object descriptor, where the objectID is the id assigned to the object by the logger and the descriptor contains all the information about the object.

The function implementing the ObjectAccessed interface is called when the log record indicates that an object has been accessed and the function implementing the ReferenceCreated interface is called when a reference is created on an object. These functions update the object descriptor or create a descriptor, if one has not been created before.

Another data structure used for making the heap model is LivenessMap, as shown in Figure 3.5. The purpose of LeakSpot is to find those leaked objects that cannot be collected by garbage collection; however, the profiling data received from the client consist of information about all the objects that are created and used during program execution, i.e, both reachable objects (live objects) and unreachable objects (garbage). Therefore, LeakSpot determines the live objects and uses the LivenessMap data structure to store the liveness information. In Section 3.5.3 we will describe the process of determining live objects.

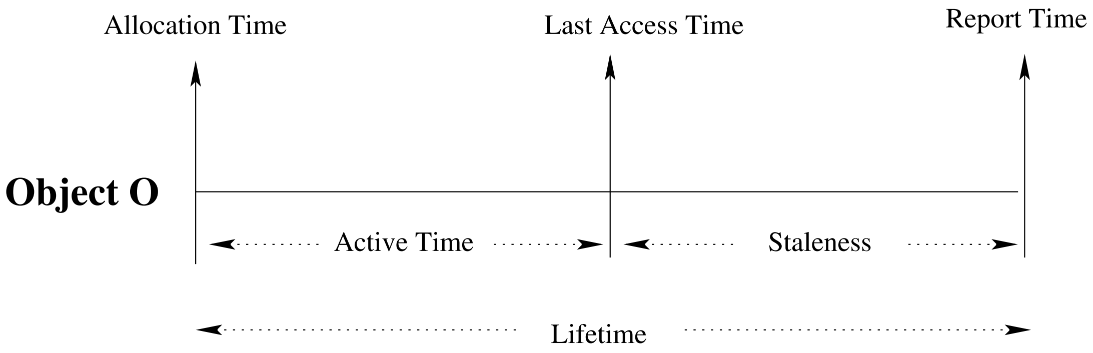


Figure 3.6: Different timing definitions related to an object

3.4.2 Determining Leaked Objects

This section describes the heuristics that LeakSpot uses to determine leaked objects and, consequently, specify those parts of the code that are allocating leaked objects or maintaining unwanted references to objects. In the following, the heuristics are described in more detail. Before going further, few definitions are provided.

Object staleness: This term refers to the length of time since the last access to an object. The staleness of an object, $Staleness(O)$, is computed as follows:

$$Staleness(O) = T_R - T_{LA(O)}$$

where T_R is the time of report generation and $T_{LA(O)}$ denotes the time of last access to the object. Time is measured by the number of allocation or access events. In other words, every call to a logging function that records allocations of objects or accesses to objects advances the time. Since web applications are interactive, using this definition for time helps reduce inaccuracy when the web application is idle [29]. Figure 3.6 visualize the staleness for an object, $Staleness(O)$, along with other timing information related to an object such as lifetime, last access time, and so on.

Collective staleness: This term describes a staleness summary for a set of objects. Inspired by the work of Novark *et al.* [68], the collective staleness is defined to be the cumulative distribution function (CDF) for the staleness of the objects. To compute the collective staleness for a set of objects, LeakSpot first computes the *relative staleness* of each object. The relative staleness for each object is computed by normalizing the staleness value for the object to the maximum staleness value for all the objects on the heap. In other words, the relative staleness of object O is computed as follows:

$$RelativeStaleness(O) = \frac{T_R - T_{LA(O)}}{MaxStaleness + 1}$$

where maximum staleness is computed as follows:

$$MaxStaleness = Maximum\,\{Staleness(O)\}\,\forall O \in Heap$$

where $Heap$ is the set of all objects on the heap according to the profiling data.

The collective staleness for a set of objects, $S_L \subset Heap$, is the CDF (Cumulative Distribution Function) of the relative staleness values for objects in the set. In other words,

$$CollectiveStaleness(S_L) = CDF\,\{RelativeStaleness(O_i)\}\,\forall O_i \in S_L$$

where

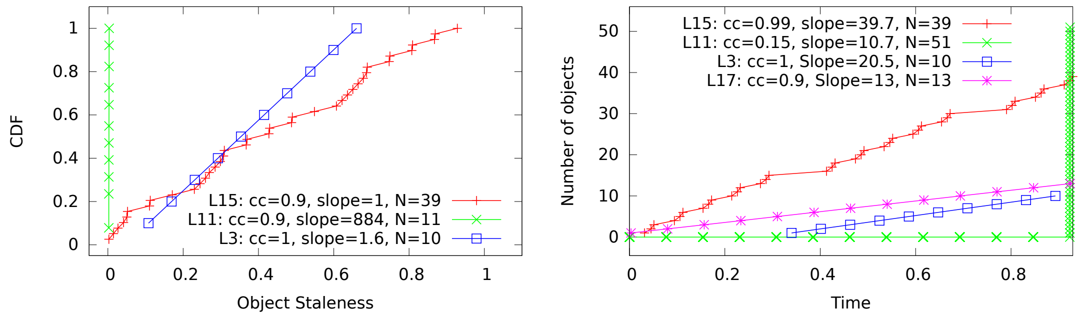
$$S_L = \{O_1, O_2, O_3, \ldots, O_i, \ldots\} \subset Heap$$

L could refer to an allocation site or reference site and, hence, S_L refers to the set of objects allocated at allocation site L or the set of objects referenced at reference site L.

Time Count: For a set of objects, it is a graph showing the number of objects added to the set over time, e.g., the number of objects allocated at an allocation site or the number of objects referenced at a reference site over time.

To find leaked objects and the leaky locations in the code, LeakSpot groups objects based on their *allocation sites* or *reference sites* and, consequently, determines whether the group of objects are leaked or not. The *SiteMap* data structure shown in Figure 3.5 is used for categorizing objects; it maps every allocation site or reference site to the set of objects allocated or referenced at that site. More specifically, LeakSpot uses the following two heuristics to determine leaked objects and leaky locations in the code:

- **Allocation-Sites Heuristic (ASH)**: This heuristic finds the locations in the program that allocate stale objects. To achieve this goal, it groups objects based on the location in the code that they are allocated, i.e., their allocation site. It then generates the collective-staleness and time-count graphs for each group of objects. If a collective-staleness graph indicates a leak, i.e., the collective staleness graph is diagonal, or if a time-count graph indicates a leak, i.e., the corresponding graph has positive slope, the corresponding group of objects are reported to be leaked and the corresponding allocation-site is reported to be leaky.



(a) Collective Staleness

(b) Time Count

Figure 3.7: Examples of collective-staleness and time-count graphs

- **Reference-Sites Heuristic (RSH)**: This heuristic categorizes objects based on the locations where a reference is created to them. It then computes the collective-staleness and time-count graphs for the set of objects to determine whether the corresponding reference site is leaky or not. In this way, LeakSpot can identify those locations in the program that accumulate objects. This heuristic is especially useful for finding objects that are no longer used but are alive because of unwanted references from library data structures. As mentioned by Pienaar *et al.* [74], data structures are one of the common causes of memory leaks in JavaScript.

Figure 3.7(a) shows the collective-staleness graph for several allocation/reference sites that are leaky or not leaky. The graph corresponding to L15, which is close to a diagonal line, denotes a leak that increases steadily over time. L15 is the reference creation point in Dojo library, line 15 in Figure 3.1. The graph corresponding to L3 indicates that $L3$ is a leaky allocation site. L3 is an allocation site in the code-load.js test case as shown in Figure 3.8.

Figure 3.7(b) shows the time-count graphs for the corresponding sites in Figure 3.7(a). As can be seen, the collective-staleness graphs and time-count graphs provide similar information. This similarity is due to the use of relative staleness values in generating the collective-staleness graphs, which provide a view of the time during the execution of an application that objects are allocated or a reference is created on them. For example, $L11$ is an allocation site in the Box2d test case where objects are not stale. From the corresponding time-count graph in Figure 3.7(b), it is clear that objects are allocated close to the end of the execution.

The only difference between the collective-staleness and time-count graphs in Figure 3.7

is the presence of allocation site $L17$ in Figure 3.7(b), which represents a leaky allocation site in the pdf.js test case of Octane benchmark, i.e., line 17 in Figure 3.14. This memory leak is due to a log buffer that is accessed for debugging purposes at the end of the execution of the pdf.js program, i.e., in the `tearDownPdfJS` function in lines 41 to 48 in Figure 3.14. The memory leak occurs because the log buffer is not emptied at the end of one run of the program. Since all the objects inside the log buffer are touched in the for loop at the end of one run, the staleness values of all the objects in the buffer are reset. Therefore, an approach based on staleness cannot be used for detecting these leaked objects.

This difference in the graphs and the shortcoming of collective-staleness graph reflects a limitation of the staleness-based heuristics [50]. For example, data structures such as arrays and hash tables occasionally touch all the objects, e.g., in a re-hash of the table, which results in false negatives in staleness-based techniques.

3.4.3 Finding Related Leaky Allocation Sites

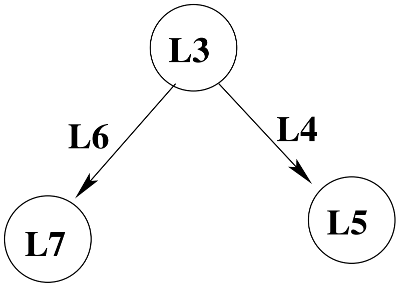
This section presents the technique LeakSpot uses for refining the list of reported leaky allocation sites. This refinement reduces the number of reported leaky allocation sites to those that are the main causes of the leaks, thus enabling a developer to fix the memory leaks faster.

Although reporting leaky allocation sites is useful, in many cases, allocation sites are related. Allocating an object, e.g., object O, may result in the allocation of many other objects. Therefore, if the allocation site of object O is reported to generate leaked objects, all the other allocation sites are reported as leaky.

To take advantage of the aforementioned relation and provide more visibility for the cause of a leak, LeakSpot makes a graph based on all the allocation sites and their corresponding relations. Every node in this graph represents a leaky allocation site and an edge from node X to node Y means that the object allocated at allocation site X has a reference to the object allocated at allocation site Y. The label on an edge indicates where in the code the reference is created, i.e, the line number in the code. Having the graph of allocation sites, LeakSpot refines allocation sites by removing nodes from the graph: a node X is removed if all the objects allocated at allocation site X are being referenced *only* by the objects allocated in one of the allocation sites inside the graph. In other words, if the reference-bearing allocation sites are outside the graph, we do not remove X. The obtained graph is visualized using visjs [9], which provides more insight to developers about the causes of leaks.

```
 1 function runClosure() {
 2 ( function() {
 3   var src = " var googS = {};
 4     googS.addDependency =
 5         function (a, b, c) {}
 6     googS.exportPath_ =
 7         function (a, b, c) {}";
 8     ...
 9     ...
10   var result = eval(src);
11   //To fix the leak
12   eval("googS = {}");
13  })();
14 }
```

(a) Simplified code of the memory leak



(b) Allocation-sites graph. LN corresponds to line number N in the code. *L*3 is the allocation site in line 3 where googS object is created, *L*5 is the allocation site in line 5 where a function is allocated, and so on.

Figure 3.8: Memory leak in the code-load test case

Figure 3.8 shows a simplified version of the memory leak in the code-load test case. It also shows the simplified allocation-site graph. As can be seen, objects allocated at allocation site L3, which is the main cause of the leak, keep objects allocated at allocation sites L7 and L5 alive; therefore L5 and L7 are reported as leaky allocation sites. After applying the above-mentioned refinement, one allocation site remains, i.e., L3, which is the main cause of the leak.

Please note that the list of reported reference sites can be refined by keeping only those sites that refer to the objects allocated in one of the refined allocation sites. This technique of refining leaky allocation sites has been possible by monitoring reference creation points as well as object modifications to know which object is creating the reference. In the case of context references, LeakSpot monitors only the points where a reference is created, and do not monitor the object function that is creating the reference. Monitoring the objects creating context references is left for future work.

3.5 Implementation

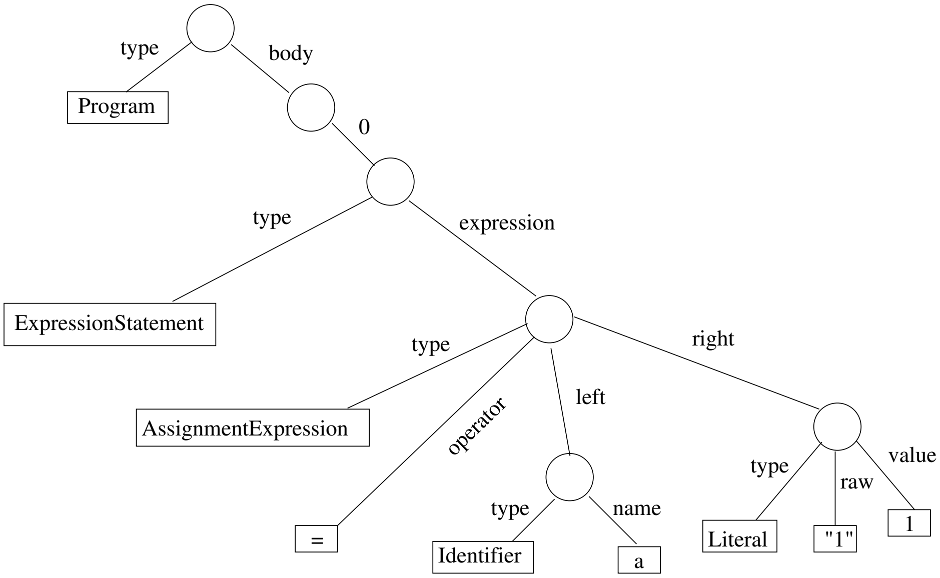
This section describes the implementation of the individual components of LeakSpot. The proxy in LeakSpot is implemented in Node.js [43]. It intercepts all the messages between the client and the server, over both HTTP and HTTPS connections; the HTTPS connections are handled by implementing a man-in-the-middle (MITM) proxy, and the SSL certificate issues are addressed by making the certificate authority known to the browser and using multi-domain certificates. The details of the proxy implementation are provided in Appendix A.

3.5.1 AST Modifications

The profiler uses the Esprima library [30] to generate the AST and uses the Escodegen library [88] to generate code from the modified AST. Figure 3.9 shows a graphical representation of the AST of a simple JavaScript statement as well as the same AST in JSON format. The generated AST is compatible with the Mozilla JavaScript parser API [66]. Table 3.1 provides the list of node types that are modified by LeakSpot, along with examples. In this table, `$` denotes the profiler library. For presentation simplicity, only one applicable instrumentation for each statement is shown. The modifications that are performed on an AST are described in the following.

Object allocations: As shown in Table 3.1, to make the code capable of monitoring object allocations at runtime, six types of AST nodes are modified: `NewExpression`, to monitor allocating objects by calling function constructors; `ArrayExpression` and `ObjectExpression` to monitor object allocations by object literals and array literals; `FunctionDeclaration` and `FunctionExpression` to monitor allocation of objects by function declaration, since in JavaScript declaring a function allocates a callable object; and `CallExpression` to handle the cases in which objects are allocated by calling specific functions such as `Object.create`.

Object accesses and modifications: Objects are accessed by reading the value of a variable or object properties. In addition, objects are modified when a value is written to the object. To monitor accesses and modifications, `CallExpression` is modified to intercept accesses to the functions, `Identifier` is modified to monitor reading of variables, and `MemberExpression` is modified to monitor when an object property or



(a) Tree representation

```
{
 "type": "Program",
  "body": [
  {
   "type": "ExpressionStatement",
   "expression": {
    "type": "AssignmentExpression",
    "operator": "=",
    "left": {
     "type": "Identifier",
     "name": "a"
    },
    "right": {
     "type": "Literal",
     "value": 1,
     "raw": "1"
    }
   }
  }
 ]
}
```

(b) JSON format

Figure 3.9: AST of a simple JavaScript statement: `a=1`

	Node Type	Statement	Modified Statement
Allocations	NewExpression	`objn = new f()`	`objn = $.logNewObject(LN, new f())`
	ObjectExpression	`objm = { }`	`objm = $.logNewObject(LN, {})`
	CallExpression	`object.create()`	`$.logNewObject(LN, object.create())`
	ArrayExpression	`obja = [ ]`	`obja = $.logNewObject(LN, [])`

	FunctionDeclaration	`function outer(){`
		`  function inner{`
		`  }`
		` }`
			`function(outer){`
			`  $.logNewFunction(LN, inner);`
			`  function inner() {`
			`  }`
			` }`

	FunctionExpression	`(function(){})()`	`( $.logNewFunction(LN, function (){}) )()`

Accesses/Modifications	Call Expression	`func( objn )`
		`objn.method()`
			`$.logFuncCall(LN, func)( objn )`
			`$.logMethodCall(LN, objn, 'func').func()`

	Identifier	`objn`	`$.logRead(LN, objn)`

	MemberExpression (get)	`objn.prop`
		`objn[i]`
		`obja[expr]`
			`$.logGetFieldDot(LN, objn, 'prop').prop)`
			`$.logGetFieldBracket(LN, objn, 'i' ) [i]`
			`$.logGetFieldBracket(LN, obja, t=expr)[t]`

	MemberExpression (put)	`objn.prop = {}`
		`objm[i] = objn`
		`obja[expr] = objn`
			`$.logPutFieldDot(LN, objn, 'prop').prop = {}`
			`$.logPutFieldBracket(LN, objm, 'i')[i] = objn`
			`$.logPutFieldBracket(LN, obja, t = expr)[t] = objn`

Reference Creations	VariableDeclaration	`var objc = objn`	`var objc = $.logVarDeclaration(LN, objn)`
	AssignmentExpression (Identifier)	`objn = objc`	`objn = $.logWrite(LN, objn, objc)`

	AssignmentExpression (MemberExpression)	`objn.prop = objc`
		`objm[i] = objn`
		`obja[expr] = objn`
			`objn.prop = $.logRightSidePutDot(LN, objc)`
			`objm[i] = $.logRightSidePutBracket(LN, objn)`
			`obja[expr] = $.logRightSidePutBracket(LN, objn)`

	CallExpression	`obja.push( objn )`	`$logMethodCall(LN, obja, 'push').push(objn)`

Table 3.1: Node types modified in LeakSpot, $ denotes the profiler library

```
var qa="createElement";
var c=J[qa]("script");
c.type="text/javascript";
c.async=!0;c.src=a;c.id=b;
var d=J.getElementsByTagName("script")[0]
```

Figure 3.10: Dynamic aliasing in the JavaScript code of Twitter

```
gbar_.S=function(a,c)
{var d=c||window.document,e=null;
d.querySelectorAll&&d.querySelector?e=d.querySelector("."+a):e=fe(a,c)[0];
return e||null};
```

Figure 3.11: Dynamic aliasing in the JavaScript code of GMail

element is being read. The nodes that are needed to monitor object modifications are listed in Table 3.1.

Reference creations: There are three types of references in the JavaScript heap: a *Property* reference created when an object is assigned as the property of another object or added to a map data structure, an *element* reference created when an object is added to an array, and a *context* reference created implicitly between a function object and a variable defined in its enclosing scope. To modify the code so as to make it possible to monitor reference creations during runtime, four types of nodes in the AST are modified: `AssignmentExpression (MemberExpression)`, to monitor the location in the program where a property or element reference is created; `AssignmentExpression-(Identifier)` and `VariableDeclaration`, to monitor those locations in the program where a context reference is created; and `CallExpression`, to handle the cases where a reference is created between two objects by calling specific functions such as `Array.push`

3.5.2 Handling Dynamic Aliases in JavaScript

Because of dynamic aliasing in JavaScript, any function call could potentially be a call to a function that allocates an object, or create references between objects. The code snippet in Figure 3.10 shows an example of dynamic aliasing in the JavaScript code of Twitter, where `J[qa]` is equivalent to `document[createElement]`. Another example of dynamic aliasing in GMail code is shown in Figure 3.11.

Logging Functions	Allocation	Access	Reference Creation	Modified	Function name and arguments
		✓			`logFuncCall(LN, func)`
	✓				`logNewFunction(LN, func)`
		✓			`logGetFieldDot(LN, object, propertyname)`
		✓			`logGetFieldBracket(LN, object, propertyname)`
		✓			`logMethodCall(LN, object, methodname)`
	✓				`logNewObject(LN, object)`
		✓		✓	`logPutFieldDot(LN, object, propertyname)`
		✓		✓	`logPutFielBracket(LN, object, propertyname)`
		✓	✓		`logRightSidePutBracket(LN, object)`
		✓	✓		`logRightSidePutDot(LN, object)`
			✓		`logRead(LN, object)`
			✓		`logVarDeclaration(LN, object)`
			✓	✓	`logWrite(LN, leftObject, rightObject)`

Table 3.2: The logging functions defined in the profiler library

Dynamic aliasing is especially problematic for cases that object allocations, accesses, or reference creations occur by calling a specific function, e.g., `object.create` which allocates an object or `Array.push` which creates a reference between objects. Because of aliasing, the Profiler treats the specific function call as a regular function call and modifies the code so that only accesses to the function object are recorded during runtime – not the allocation, access, or reference creation that occurs by calling the function.

To address this case, the JavaScript functions of interest are wrapped [80]. Wrapping a function means another function (the wrapper) calls the original function (the wrapped one), rather than the original function getting called directly. Therefore, it is possible to execute some code right before and after calling the original function, including logging the information of interest. The wrapped functions are sent to the client browser and are executed in the browser before the application is executed. Therefore, any call to the original function will be redirected to the newly wrapped function. Figure 3.12 shows an example of how a function is wrapped.

In the current implementation of LeakSpot, we chose to wrap two functions, as shown

```
function WrapDomFunction(func, extra) {
 return function() {
  //before function execution code goes here
  var wrappedfunction=func.apply(this, arguments);
  //after function execution code goes here
  return wrappedfunction;
 }
}
document.createElement = WrapDomCreateElement(document.createElement)
```

Figure 3.12: An example of how a function is wrapped

in Table 3.1, i.e., `Object.create` which is a factory method, and `array.push` which creates a reference between an array and the element inserted into it. To get the location in the code where a function is called, the AST modifications to monitor function calls are still required.

3.5.3 Extracting Live Objects

To make the heap model in Leak Reporter using only live objects, LeakSpot needs to determine which objects are live. To do so, a heap snapshot is collected manually using the facilities provided in Chrome Developer Tools [34]. The heap snapshot is collected after performing a full garbage collection; therefore, the unreachable objects (dead objects) are not on the heap. For each object in the profiling data, Leak Reporter checks whether the object is on the heap. Then, it creates a data structure that maps every object id (the id assigned to the object by the logger) to a boolean value that indicates whether the object is on the heap (live). *Please note that the heap-snapshot is collected only once at the time of report generation.* As an alternative approach, determining the list of live objects could be done by implementing a modified version of reference-counting garbage collection. Implementing this technique requires more instrumentations to make sure that all object references are monitored and recorded at runtime, including context references. It also requires a smart reference-counting that can handle the shortcoming of reference counting garbage collection, i.e., when two object are referencing to each other. We leave implementing this technique as future work.

3.6 Evaluation

This section aims to answer the following questions:

1. What is the accuracy of LeakSpot?

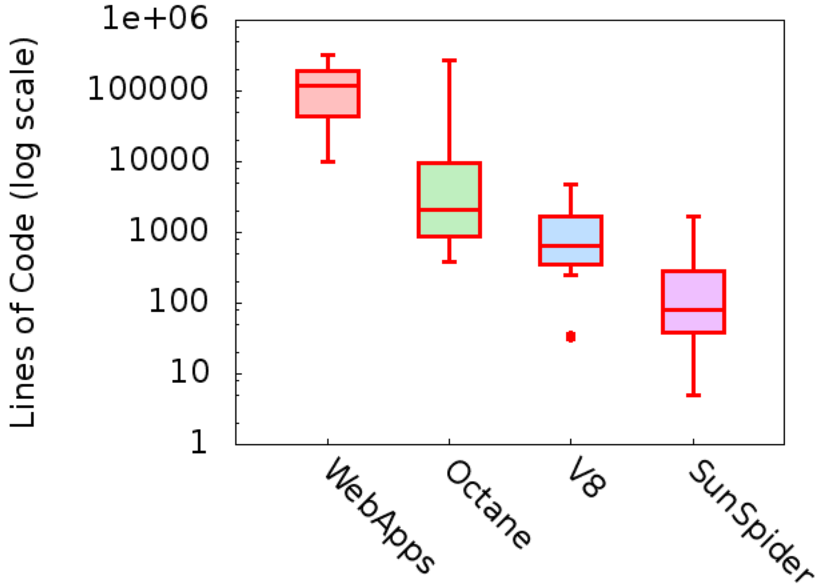


Figure 3.13: Complexity of JavaScript benchmarks and web applications

2. What is the effectiveness of LeakSpot in finding and fixing memory leaks?

3. What is the performance and computational overhead of LeakSpot on the client?

3.6.1 Experimental Setup

To perform the experiments, three machines are used. One machine is set up as the web server and used only when studying JavaScript benchmarks; another machine is set up as a proxy running LeakSpot; and a third machine is used to run he client browser and is configured to use the proxy. The proxy machine has a 2.10GHz CPU, with 12 GB of RAM; the client machine has a 800MHz CPU, with 2 GB of RAM; and the web server has a 2.40 GHz CPU, with 4GB of RAM and is used when running experiments on benchmarks. We used Google Chrome 36.0.1985.125 for running the web applications under test. To avoid the memory leaks caused by inline cache or optimized code [47] and to avoid the false report of increasing memory in the case of incremental marking [12], Chrome is run with the following command line flags:

–js-flags="–nouse-ic –nocrankshaft –gc-global –noincremental_marking "

To perform the evaluation on JavaScript benchmarks, Octane benchmark [70] is chosen because it has higher complexity than other well-known JavaScript benchmarks such as SunSpider [92] and V8 [40]. Figure 3.13 compares the complexity of different test cases in JavaScript benchmarks and web applications in terms of lines of code. Table 3.3 shows some statistics about the Octane benchmark including lines of code, size of heap snapshot, and size of profiling data.

3.6.2 Accuracy of LeakSpot

To study the accuracy of LeakSpot, it is necessary to have cases that are *known* to be leaks and then see how LeakSpot performs in those cases. To obtain such known cases, each test case in Octane benchmark is executed repetitively to see whether it is leaky[2]. To find whether a repetitive execution of a program generates leaked objects, the *heap-size* graph is produced, which shows the size of live objects on the heap of an application over time. The time is measured in the size of objects allocated on the heap, similar to the approaches in previous research [1, 17]. To make sure that each point on the heap graph is the size of live objects at that time, garbage collection is performed before measuring the size of live objects. In a repetitive scenario, an increasing trend on the heap-size graph indicates that all the allocated objects have not been released.

To identify the leaky test cases, all of the test cases are run repetitively. The results are shown in Table 3.4; the *Leaky* column highlights the test cases that are leaky. To detect leaked objects and leaky sites in each test case, the collective-staleness and time-count graphs for all the allocation sites and reference sites in the test case are generated. To find whether a collective-staleness graph or time-count graph indicates a leak, a line is fitted over the data using linear regression. A time-count graph indicates the existence of a memory leak if the correlation coefficient of the line fitted to the data is larger than 0.7 and the slope is positive. A positive slope indicates that the number of objects over time are going up. If the set of actions are repetitive, even a slight increase in number of objects denotes memory leak. A collective-staleness graph indicates a leak if it resembles a diagonal, e.g., it is similar to the graphs of $L3$ and $L15$ in Figure 3.7(a), and the correlation coefficient of the line fitted to the data is larger than 0.7. The correlation coefficient is a threshold value that should be determined based on the desired level of sensitivity.

As shown in Table 3.4, for the non-leaky test cases, LeakSpot does not report any allocation or reference sites, as expected; but for the two leaky test cases, it reports a

[2] This method is chosen instead of the less strong method of injecting known leaks into the code of a test case.

Test Case	Lines of Code	Heap Snapshot Sizes		Profiling Data Sizes
Benchmark		Unmodified	Modified	
box2d	8508	5.2 MB	6.1 MB	125 MB
code-load	1485	4.1 MB	4.7 MB	190 KB
crypto	1665	3.6 MB	3.9 MB	60 MB
deltablue	858	3.6 MB	3.8 MB	6.4 MB
earley-boyer	4681	4 MB	4.5 MB	107 MB
gbemu	11098	4.2 MB	4.7 MB	358 MB
mandreel	277373	7.9 MB	10.1 MB	2 GB
navier-stokes	387	3.5 MB	4 MB	68 MB
pdf.js	33028	9.1 MB	13.9 MB	85 MB
raytrace	894	3.6 MB	3.8 MB	21 MB
regexp	1766	3.6 MB	4.2 MB	27 MB
richards	502	3.5 MB	4 MB	4.4 MB
splay	388	3.5 MB	3.7 MB	69 MB
typescript	25842	4.8 MB	5.6 MB	1.3 GB
zlib	2494	3.6 MB	4.1 MB	5 GB
Common Code	9791	–	–	–

Table 3.3: Statistics about Octane benchmark

Test Case	Leaky	Collective Staleness			Time Count	
		Allocation Sites	Reference Sites	Refined Allocation Sites	Allocation Sites	Reference Sites
box2d	No	0	0	0	0	0
code-load	Yes	**41**	**42**	1	**41**	**42**
crypto	No	0	0	0	0	0
deltablue	No	0	0	0	0	0
earley-boyer	No	0	0	0	0	0
gbemu	No	0	0	0	0	0
mandreel	No	0	0	0	0	0
navier-stokes	No	0	0	0	0	0
pdfjs	Yes	**0**	**0**	0	**1**	**2**
raytrace	No	0	0	0	0	0
regexp	No	0	0	0	0	0
richards	No	0	0	0	0	0
splay	No	0	0	0	0	0
typescript	No	0	0	0	0	0
zlib	No	0	0	0	0	0

Table 3.4: Results of evaluating LeakSpot on test cases in Octane benchmark

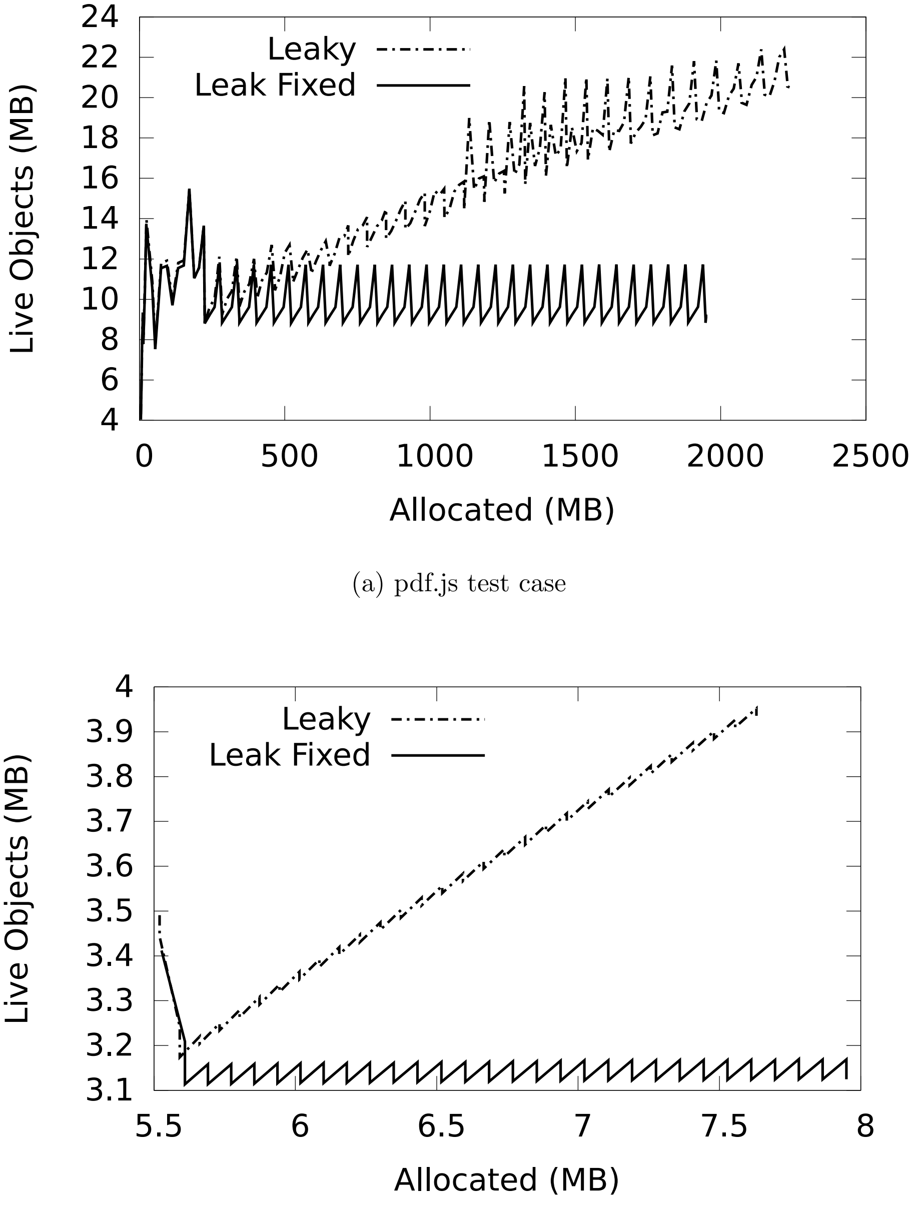
number of allocation sites or reference sites as leaky. Figure 3.15(a) and Figure 3.15(b) show the heap-size graphs for the two leaky test cases in Octane benchmark, i.e., code-load and pdf.js, before and after fixing the leaks.

For the pdf.js test case, LeakSpot reports one leaky allocation site and two leaky reference sites. Looking at the code of the pdf.js test cases, the one leaky allocation site based on time-count graph findings, is site $L17$ in Figure 3.14. Since LeakSpot monitors references created to an object and modifications to the object, it is possible to easily find where a reference is created and what object created the reference. Two references are created on the objects allocated at $L17$; one in line 38 from a data structure, `canvas_logs`, and one in line 17 from `this` object.

For the code-load test case, LeakSpot reports 41 allocation sites as leaky, which reduces to one allocation site after refining the reported allocation sites. Figure 3.8(a) shows a simplified version of the parts of the code in the code-load test case that cause the memory leak. The leaky allocation site that is causing the leak, $L3$, denotes the object allocated

```
10  var canvas_logs = [];
11  var PdfJS_window = Object.create(this);
12  function PdfJS_windowInstall(name, x) {
13      Object.defineProperty(PdfJS_window,
14          name, {value: x});
15  }
16  PdfJS_windowInstall("Context", function() {
17      this.__log__ = [];   //L17
18      this.save = function() {
19          this.__log__.push("save","\n");
20      }
21  })
22  PdfJS_windowInstall("Canvas", function() {
23      this.getContext = function() {
24          return new PdfJS_window.Context();
25      }
26  });
27   PdfJS_windowInstall("document", {
28      getElementById : function(name) {
29         if (name === "canvas") {
30           return new PdfJS_window.Canvas();
31      } } });
32  function runPdfJS() {
33    var canvas = PdfJS_window.document.
34                    getElementById('canvas');
35    var context = canvas.getContext('2d');
36
37    //reference creation point
38    canvas_logs.push(context.__log__);
39  }
40  function tearDownPdfJS() {
41      for (var i = 0; i < canvas_logs.length; ++i) {
42      var log_length = canvas_logs[i].length;
43      ...
44      }
45    }
46    delete this.PDFJS;
47    delete this.PdfJS_window;
48    canvas_logs = []; // To fix the leak
49  }
```

Figure 3.14: Simplified leaky code in the pdf.js test case



(a) pdf.js test case

(b) code-load test case

Figure 3.15: Heap-size graphs for the two leaky test cases in Octane benchmark

at line three and put into `googS` variable. The memory leak is fixed by setting the value of the `googS` object to an empty object at the end of the code of code-load test case.

3.6.3 Case Studies on Open-Source Web Applications

In the following experiments, the effectiveness of LeakSpot in guiding the developer in fixing the leaks quickly and easily is studied. The experiments are performed on Todo web applications [93] developed using three different libraries, i.e., Dojo [18], AngularJS [33], and YUI [45]. These Todo applications are examples of single-page web applications [59] that LeakSpot targets. Table 3.5 shows some statistics about the open-source web applications under study including lines of code, size of heap snapshot, and size of profiling data.

To expedite the effects of the memory leaks in the web applications, a set of actions is performed repetitively on the page. In the Todo web applications under study, a series of tasks are added and removed to/from the todo list, repetitively. Table 3.6 highlights the leaky applications as well as the results of evaluating LeakSpot on these open-source web applications.

LeakSpot does not report any leak for the Todo-AngularJS application, as expected, since the heap-size graph corresponding to this application does not indicate a leak. For the other two web applications, LeakSpot reports a number of the allocation sites and reference sites as leaky. Table 3.6 also presents the number of *Refined Allocation Sites* for each web application, i.e., the number of leaky allocation sites after applying the refinement technique explained in Section 3.4.3. As can be seen, this technique reduces the number of reported allocation sites significantly and empowers developers in fixing the leak. The time-count graph numbers are higher than those from collective-staleness graphs due to the false-negative issues inherent in a staleness-based techniques, as discussed in Section 3.4.2.

Figure 3.17 shows the heap-size graphs of the leaky web applications before and after fixing the leaks. The small vertical lines in each graph indicate when a garbage collection is performed, i.e., after addition and removal of five tasks to/from the todo list. For the Todo-Dojo application, LeakSpot reported 10 leaky allocation sites and 11 leaky reference sites after refinement. One of the reference sites, i.e., line 15 in Figure 3.1 was causing the leak as explained before.

Using LeakSpot, it was also possible to find and fix a memory leak in the Todo-YUI application. After refining the leaky allocation sites, LeakSpot reported 12 allocation sites. One of them was a leaky allocation site in the developer code and directed us to fix the memory leak. Figure 3.18 shows a simplified version of the Todo-YUI code. The part of

WebApp	Lines of Code	Heap Snapshots Sizes		Profiling Data Sizes
		Unmodified	Modified	
TodoAngular	10211	4.8 MB	5.2 MB	146K
TodoDojo	44206	5.9 MB	7.9 MB	11 MB
TodoYUI	44206	6.5 MB	9.1 MB	11 MB

Table 3.5: Statistics about the open-source web applications

WebApp	Leaky	Collective Staleness			Time Count	
		Allocation Sites	Reference Sites	Refined Allocation Sites	Allocation Sites	Reference Sites
TodoAngular	No	0	0	0	0	0
TodoDojo	Yes	52	33	10	52	33
TodoYUI	Yes	47	66	12	47	66

Table 3.6: Results of evaluating LeakSpot on the open-source web applications

the code that is causing the memory leak and the code added to fix the memory leak are highlighted. In this application, whenever new todo tasks are added, changed, or removed, the program goes through all the items in the todo list and assigns a view object to each item (line 56 in Figure 3.18) without properly removing the previously added view object, which causes the leak. Figure 3.16 shows the simplified heap view of the memory leak. As shown in the heap graph, unwanted references to the view objects have not been removed because of programmer error.

In the open-source web applications studied in this section, it was possible to easily fix the leaks for two reasons. First, the refinement technique reduced the number of leaky allocation sites to a small number of allocation site to investigate. Second, knowing the code locations where references are created to the objects sped up the process of fixing the leaks and saved considerable developer time.

3.6.4 Case Studies on Large Web Applications

The following experiments demonstrate the existence of memory leaks in large and popular web applications. In addition, the results of applying LeakSpot on the data collected during the execution of web applications is presented. The web applications under study

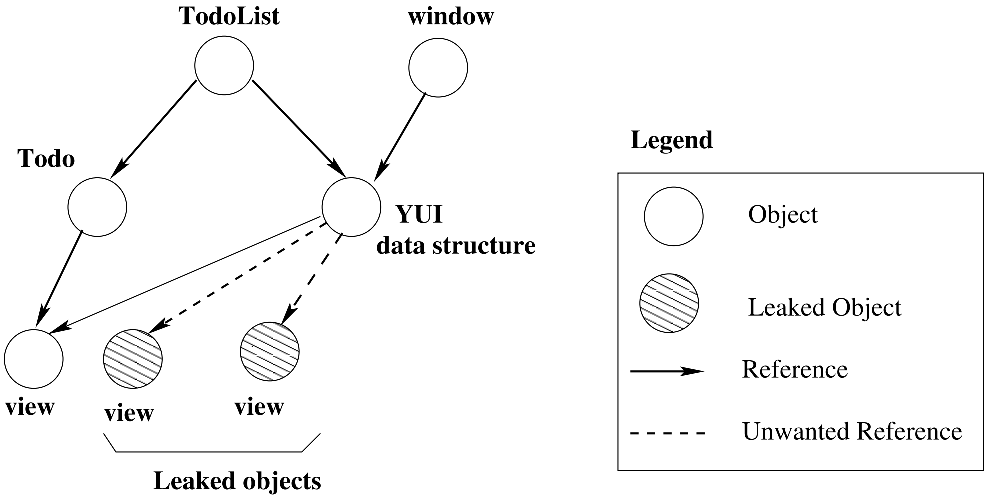


Figure 3.16: Simplified heap view of the memory leak in Todo-YUI

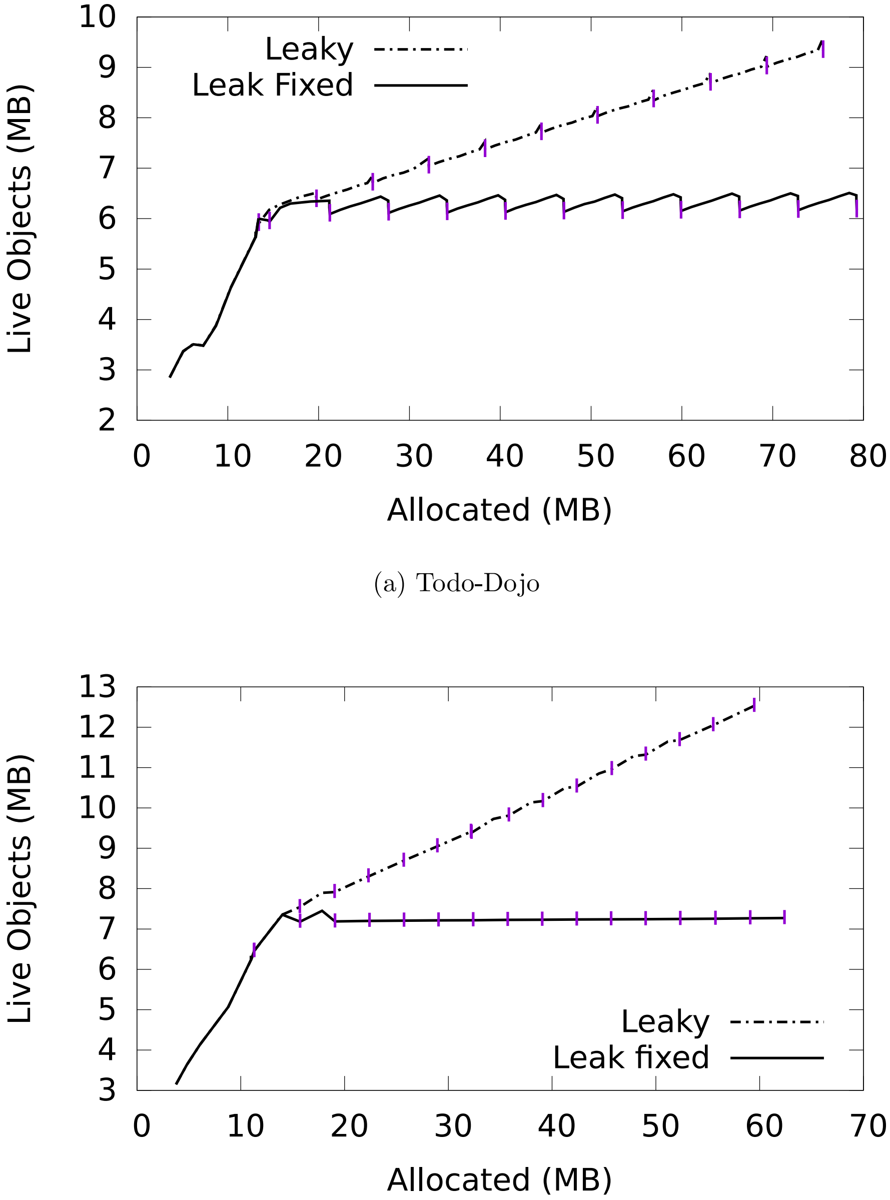
WebApp	Lines of Code	Heap Snapshots Sizes		Profiling Data Sizes
		Unmodified	Modified	
GMail	325906	46	129	205
Calendar	111755	20	40	100
YMail	191892	27	39	378
FaceBook	128082	29	71	838

Table 3.7: Statistics about large web applications

are chosen based on their popularity according to the Alexa [32] web site, and also based on their development technology. LeakSpot targets single page web applications [59] and, therefore, those web applications that fully or partially follow this application model are chosen. Table 3.7 shows some statistics about the web applications under study.

For every web application, a set of repetitive actions is done repetitively that may or may not generate leaked objects. Table 3.8 shows the list of actions performed on web applications under study and Figure 3.19 shows the corresponding heap-size graphs. The small vertical lines in the figures indicate when a garbage collection is performed.

Table 3.9 shows the results of applying heuristics. LeakSpot does not report any leaks for Calendar, as expected, since there is not an increasing trend in its corresponding heap-size graph (Figure 3.19). For the other web applications, LeakSpot reports a number of



(a) Todo-Dojo

(b) Todo-YUI

Figure 3.17: Heap-size graphs for the leaky open-source web applications

```
YUI.add('todo-app', function (Y) {

  var TodoApp;
  var TodoList = Y.TodoMVC.TodoList;
  var TodoView = Y.TodoMVC.TodoView;

  TodoApp = Y.Base.create('todoApp', Y.App, [], {

    initializer: function () {
      this.set('todoList', new TodoList());
      var list = this.get('todoList');
      list.after(['add', 'remove', 'reset', 'todo:completedChange'],
        this.render, this);
    },

    render: function () {
      var todoList = this.get('todoList');
      var completed = todoList.completed().size();
      var remaining = todoList.remaining().size();
      var footer = this.get('footer');

      footer.one('#filters li a').removeClass('selected');
      footer.all('#filters li a')
      .filter('[href="#/' + (this.get('filter') || '') + '"]').addClass('selected');
      this.addViews();
    },

    addViews: function () {
      var models;
      var fragment = Y.one(Y.config.doc.createDocumentFragment());
      var todoList = this.get('todoList');

      switch (this.get('filter')) {
        case 'active':
          models = todoList.remaining();
          break;
        case 'completed':
          models = todoList.completed();
          break;
        default:
          models = todoList;
          break;
      }
      models.each(function (model) {

        //The following line is causing the leak
        var view = new TodoView({model: model});

        fragment.append(view.render().get('container'));
      });
      //To fix the leak add the following line
      this.get('container').one('#todo-list').destroy({recursivePurge: true});

      this.get('container').one('#todo-list').setContent(fragment);
    }
  });
})
```

Figure 3.18: Simplified code of the Todo-YUI web application

WebApp	**Repetitive Actions**
GMail (Google Mail)	Opening and closing the compose window.
Calendar (Google Calendar)	Creating and deleting events.
YMail (Yahoo Mail)	Opening and closing the compose window.
Facebook	Opening and closing an image in the news feed.
Todo	Adding a series of tasks and removing them repetitively.

Table 3.8: Repetitive actions performed on the web applications

		Collective Staleness			Time Count	
WebApp	Leaky	Allocation Sites	Reference Sites	Refined Allocation Sites	Allocation Sites	Reference Sites
GMail	Yes	19	20	6	20	35
Calendar	No	0	0	0	0	0
YMail	Yes	117	120	47	117	84
FaceBook	Yes	97	107	30	97	107

Table 3.9: Results of evaluating LeakSpot on large web applications

allocation and reference sites as leaky.

As can be seen, the technique used to refine the reported allocation sites reduced the number of leaky allocation sites significantly. For GMail, it reduced the number of reported allocation sites to six, and for Facebook and Yahoo, it reduced the number of leaky allocation site to less than one third of the originally reported numbers. These experiments demonstrate the existence of memory leaks in large and popular web applications and LeakSpot's potential to find the cause of such memory leaks. We could not fix the leak since the obtained code is obfuscated; however, the fact that LeakSpot was able to guide us in fixing the memory leaks in Octane benchmark and open-source web applications and the low number of allocation sites strongly suggests that it would be quick to check the relevant sites and fix the leaks. In addition, evaluation of LeakSpot on large and complex web applications demonstrates the scalability of LeakSpot.

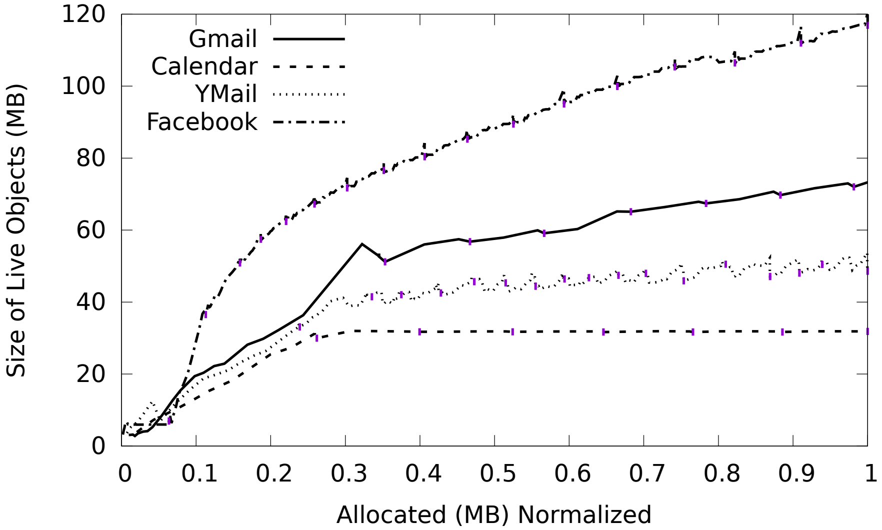


Figure 3.19: Heap-size graph for large web applications

3.6.5 Measuring Performance Overhead

This section studies the performance overhead of LeakSpot. The purpose of measuring overhead is to highlight any bottleneck in performance. Measuring the overhead of different components of LeakSpot can guide us in choosing the optimizations required to reduce the overhead. Since LeakSpot is designed to work during development mode, we are not worried about the overhead; however, the design decisions of LeakSpot make it possible to use LeakSpot in an online setting (beyond the development stage), if the overhead is reduced.

Loading and running the modified version of an application is slower compared to using the original one. There are several factors affecting the performance:

- **Modification Overhead:** LeakSpot makes loading of a page slower since it modifies the code inside a proxy before sending it to the client, as shown in Figure 3.4.
- **Communication Overhead:** This overhead is caused by the communication between the Logger and the proxy to transfer profiling data.
- **CPU Overhead**: Running instrumented code is more CPU-intensive compared to unmodified code.

To find the bottlenecks in performance overhead, the load time is measured in different configurations:

1. *Baseline*: There is no proxy in place.

2. *Unmodified*: The code goes through proxy and all modifications are applied, but the *unmodified* code is sent to the client. The purpose of this configuration is to measure modification overhead.

3. *Modified*: The modified code is sent to the client and executed in the browser but the log records are not buffered and are not sent to the proxy. The purpose of having this configuration is to measure the CPU overhead associated with executing the instrumented code in the browser.

4. *Full*: The modified code is executed in the client browser and the profiling data are sent back to the proxy. This configuration highlights the communication overhead compared to the previous configuration.

Since the page load time is a key performance metric for web applications [102], the overhead is studied by measuring the load time of web applications in different configurations. Moreover, while the use of LeakSpot results in a significant delay in the loading of a page, the response time of an application during execution is not degraded substantially. This is true for interactive single-page web applications that LeakSpot targets, where a large chunk of JavaScript is loaded into the browser, while the JavaScript code that is executed in response to user events is short. The load time is measured by using the Navigation Timing API [65]. A page load is complete when the load event is fired in the browser. Since a web page may consist of several iframes, we take the maximum time to load the iframes as the load time of the the page. We did every experiment ten times and used WebDriver [84] in Java to automate the process of repeatedly loading a page.

Figure 3.20 shows the load times for the web applications under study. The load-time value for each web application is normalized to its load time value in Full configuration. As can be seen, a jump in the load times happens between the case of Baseline and Unmodified. This jump in load times highlights the modification overhead, including having a proxy and all the parsing and code generation involved. This overhead is not inherent to the approach used in LeakSpot. There are many optimizations that can be done to remove this overhead, such as using a more powerful machine as the proxy or using a multi-threaded implementation for the proxy. In addition, LeakSpot is intended to be used by system developers, who have access to the server side, and therefore can send instrumented code and profiler library to the client directly, which eliminates modification overhead. In this work, the use of the proxy on the client side provides the opportunity to evaluate LeakSpot on common web applications that are not open source.

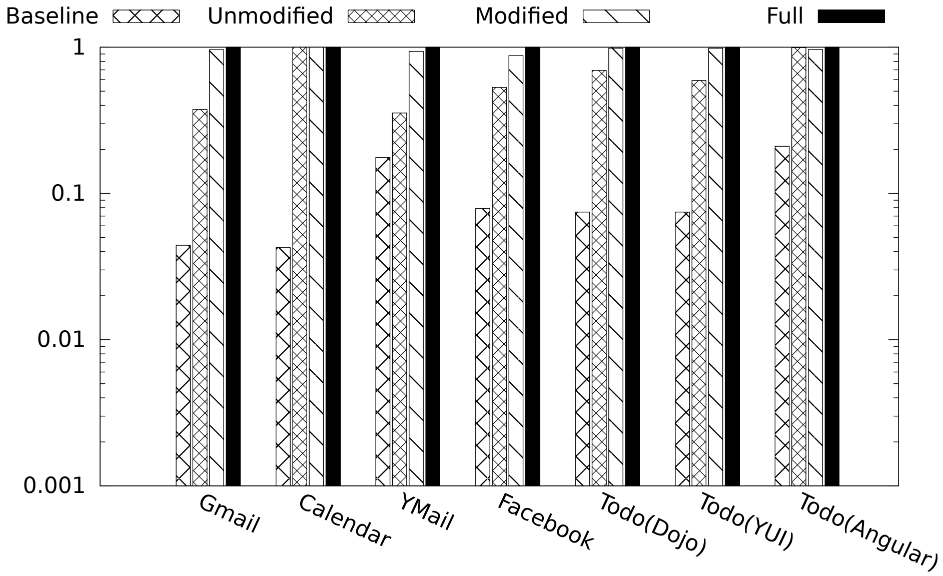


Figure 3.20: Load time of Web applications

The difference in load time between Unmodified and Modified configurations demonstrates the CPU overhead caused by the code executed during the page load. This overhead can be optimized in different ways. The data shown in Figure 3.20 reflects the case where all the JavaScript code included in a page is modified, including the application code and library code; however, modifying all the code is not always required. For example, for the Todo-YUI application, we controlled the proxy so that only the developer code is modified – not the code of the YUI library. Loading the page in this case and measuring overheads, we did not notice any difference in the load time between Modified and Unmodified configurations. It is worth mentioning that the reported leaky allocation site in the developer code was enough for detecting the leaks and finding the fix. We found similar results when repeating this experiment for Todo-Dojo and Todo-Angular. This experiment could not be repeated for the applications that are not open-source, but similar results are expected.

In addition, the source-level instrumentation technique used here allows the application of many optimizations without reducing the accuracy of the leak detection. One optimization that can be applied to reduce CPU overhead is to modify the code partially and use the redeployability [51] characteristics of web applications to send different code to different users. This is one of the advantages of web applications which allows us to send different versions of the application to different users, where in every version a different subset of the code is instrumented. In this way, the overhead will be distributed among users, and the overhead experienced by each user will be negligible. To automatically select those parts of the code that are related and dependent, program slicing [23, 54] techniques can be used. These kinds of code-initiated optimizations are not possible with previous approaches based on dynamic analysis techniques that collect runtime data of an application by instrumenting the Java virtual machine, like SWAT [29] and Bell [8].

Comparing the load times of applications in Full and Modified configurations highlights the communication overhead. Since the communication between a client and the proxy is done asynchronously, the communication overhead due to transfer of data between the browser and the proxy does not have much impact on the performance of an application. The more code is executed during load time, the larger the size of profiling data, and so the higher the effect of communication overhead. For example, in the Calendar application, the amount of profiling data is lower than the other popular applications and so is the effect of communication overhead. Table 3.7 shows the size of profiling data during the load time.

In addition to the above mentioned overheads, there is memory overhead caused by adding a unique id to every object and loading the library code into the browser. Table 3.7 show the size of heap snapshot of each web application, once measured when the code is not modified, and once measured when the code is modified. The above-mentioned optimization techniques are effective in reducing memory overhead, as well.

3.7 Discussion and Limitations

LeakSpot modifies object allocations, accesses, and references as specified by the third edition of the ECMAScript language specifications [22]. It also supports functionalities for object allocation specified in the fifth edition, such as `Object.create`. While LeakSpot modifies the code to make it possible to monitor activities on objects according to JavaScript standards, it can be extended to monitor object allocations, accesses, and reference creations using the custom allocation methods in the libraries. For example, if there are functions defined in the library for object allocation, the instrumentation in LeakSpot could be changed accordingly to monitor this new method of allocation. In addition, LeakSpot can be easily extended to collect more information about the objects, such as special functionalities for setting and getting object properties, as specified in the 5th edition of ECMAScript.

The heap of a web application is composed of JavaScript objects and DOM objects. LeakSpot focuses on finding JavaScript objects that are leaked. Since JavaScript objects and DOM objects refer to each other, finding leaked JavaScript objects also results indirectly in the removal of many leaked DOM objects. Both examples of memory leaks in the Todo-Dojo and Todo-YUI web applications caused many DOM objects to be leaked, as well. To monitor DOM object allocations, accesses, and reference creations, LeakSpot can be extended by wrapping DOM APIs [64]. Appendix B discusses the differences between DOM and JavaScript objects and lists some of the functions that need to be wrapped. Full modification and wrapping of all the functions in DOM APIs is an engineering effort.

To find the leaky allocation sites, LeakSpot computes the collective-staleness and time-count graphs for all the allocation/reference sites in the application. LeakSpot shows these graphs to the developer, who can go through them to get an idea of the behaviour of different sites. In the case of large applications, where there are large numbers of allocation sites, to make the developer's task easier, LeakSpot fits a line over the data to find the problematic allocation or reference sites automatically using linear regression. The threshold values are chosen so that the graph of the data represents a trend in the data statistically. As mentioned in [75], a threshold value larger than 0.7 indicates a strong linear relationship. The developer can set this value based on the desired level of sensitivity.

The Logger, in LeakSpot, assigns an id to the objects by adding the id as the property of objects. The property is named `__objectID__`; therefore, if the object already has a property with this name it may result in errors. It is easy to change the name on the fly if it has already been used by the application code. In addition, although the id is added as a non-enumerable property to make it invisible during execution, it can be still visible. For example, the `Object.getOwnPropertyNames` function returns all property names irrespective of their iterable nature [52]. To prevent such a problem, if the application code happens to use the `Object.getOwnPropertyNames` function, this function could be wrapped to exclude reporting any property with the specific name `__objectID__`. It is worth mentioning that the property name used currently in LeakSpot, i.e., `__objectID__`, has not introduced any problem in our experiments.

As discussed in Section 3.4.2, the staleness-based technique used in LeakSpot may report false negative results. To overcome this shortcoming, which is inherent in staleness-based heuristics, LeakSpot also reports time-count graphs for the allocation/reference sites. This technique of finding the leaky allocation sites or reference sites by tracking the number of objects over time can be used if the increasing trends in the the graphs are not the expected behaviour of the program. For instance, in the case of a repetitively executed program one would not expect to see an increasing trend in the number of objects. If there is an increasing trend there is a leak.

Since LeakSpot uses staleness and growth in number of live objects, it may result in false positives and report cases that are not real memory leaks, but could be sources of memory inefficiency. One example occurs when there is a cache data structure in the application or in the library used by the application, but there is no limit on the cache, resulting in memory bloat in long-running applications [109].

The techniques used in LeakSpot does not introduce additional false positives. First, the staleness-based method precisely identifies memory not being used, since LeakSpot monitors all accesses to the objects, unlike the approach by Hauswirth *et al.* [29] that uses adaptive profiling to reduce overhead, which, in turn, decreases the accuracy of memory leak detection. Second, since LeakSpot combines staleness information with allocation or reference sites, it significantly reduces the chances of false positives. Reporting objects based only on staleness values, i.e., reporting an object as a leak if the staleness value is larger than a threshold value, produces significant false positive results since many objects are allocated at the beginning of the program that are not used but are needed, such as the objects used for presentation and visualization of the application [107].

As discussed by Bond *et al.* [8], one of the disadvantages of memory leak detection based on per-object site data, is that the calling context is limited to the inlined portion and so may not be enough to understand the behaviour of the code causing the leak. Bond *et al.*mentioned that efficiently maintaining and representing dynamic calling contexts is an unsolved problem. Although LeakSpot collects per-object site data, it provides more context about the objects than the approach used by Bond *et al.*. The profiling data contains a history of all the references and all the accesses made to the object, which provides more data about the calling context compared to just monitoring allocation sites and the last-use site method used by Bond *et al.*. In addition, the calling context can be more specific by taking advantage of the approaches proposed by Novark *et al.* [67], i.e., identifying the sites with bounded context sensitivity, e.g., the last four functions on the call stack.

In this work, we have focused on client-side web development since we are inspired by the problems in the client side of web applications; however, memory leaks are an even bigger problem for server-side JavaScript, e.g., running on Node.js [43], as these programs often implement long-running servers [28]. Especially in the Node.js community, people have for a long time recognized the memory leak issue and commercial services exist to help developers gain insight into the behaviour of their JavaScript server program [5, 44]. We think that with slight modifications, LeakSpot can be used in these environments. The fact that the instrumentation is done by modifying the source code makes it portable (among different browsers) and also usable in different environments (like Node.js). Currently, our profiler library works for Node.js applications. We have tested the test cases in Octane benchmark in the Node.js environment as well as in the browser. In terms of memory leaks, the results are the same. Evaluating LeakSpot on the server-side applications can be a direction for future work.

3.8 Conclusion

A memory leak, which is the existence of unused objects on the heap of an application, is prevalent in JavaScript applications. In this chapter, we introduced LeakSpot, a tool for detecting and diagnosing memory leaks. LeakSpot works by modifying the code in a browser-agnostic way to monitor object allocations, accesses, and the references created to the objects. LeakSpot reports allocation sites and reference sites that are generating the leaks. In addition, it helps developers in fixing the leaks by refining allocation sites and reporting all the code locations that create references to the objects. LeakSpot is shown to be effective in finding and fixing memory leaks in JavaScript benchmark and open source web applications. Using LeakSpot, we were able to find and fix two leaks in Octane JavaScript benchmark and two leaks in open-source web applications. Our results show that the source-level instrumentation along with the redeployability characteristics of web applications provide the opportunity to apply many optimizations that make it possible to use LeakSpot beyond the development stage. In future work, we intend to apply the discussed optimizations to reduce LeakSpot overhead. In addition, we are planning to work on heuristics, specifically those that are aligned for JavaScript applications, to reduce the imprecision associated with staleness-based heuristics.

Chapter 4

Runtime Management of Memory Leaks

This chapter presents an approach for managing memory leaks at runtime. Avoiding the leaks at runtime requires fixing them during development; they cannot be fixed on a running system. Therefore, the goal is to improve memory efficiency of web applications by making them capable of automatically recovering themselves from errors due to memory leaks. Doing so means more memory is available to the page, which results in better performance and increased time-to-failure for the specific page under study and the whole system. This result is achieved by applying recovery actions to remove the errors in the system, permanently or temporarily.

This approach is implemented in a prototype tool called MemRed. The architecture of MemRed is shown in Figure 4.1. It consists of three main components: monitoring, error detection, and recovery. MemRed monitors the web application running in a browser tab, to get data about memory usage of the application. Then, it analyzes the collected data to find whether it indicates an error in the system. Finally, if the system is detected to be in an error state, the system takes proper actions to return the system to a healthy state and remove the effects of errors. In the following, we describe each component in more detail.

4.1 Monitoring

To make a system capable of automatically removing errors, it is necessary to continuously monitor its components to find whether there is an error in the system. The monitoring

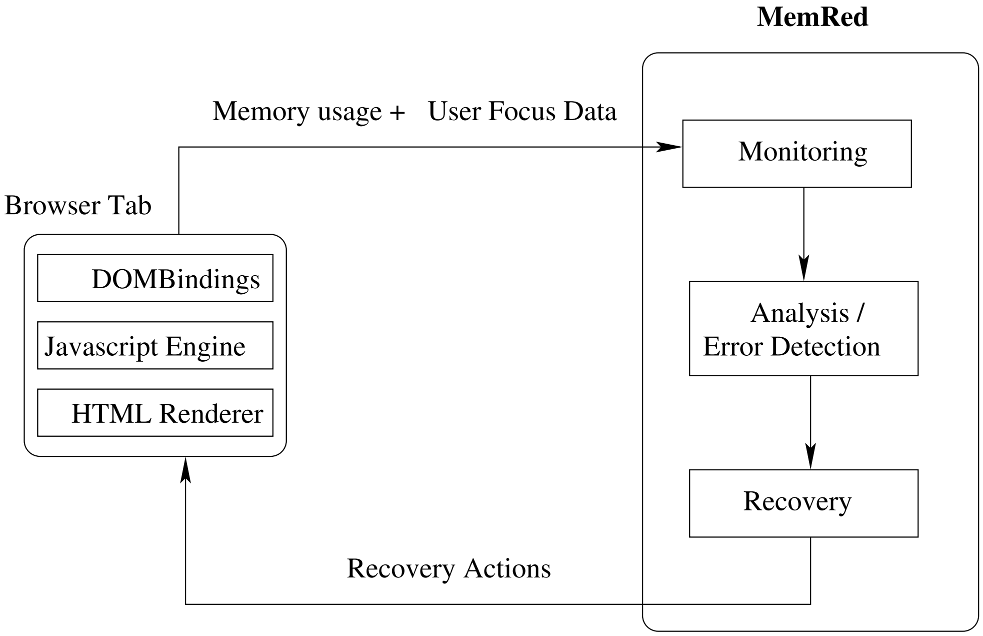


Figure 4.1: MemRed Architecture

provides useful data about system behaviour that will be analyzed by the analysis component to predict failures. The monitoring component periodically collects data about two different aspects of a web application. First, it collects memory usage data, which will be used by the analysis component to detect any signs of excessive memory usage in the system. Second, it monitors the status of the tab with respect to user visibility and focus. This data is used by the recovery component to apply recovery actions in such a way that user interruption is minimized. In summary, the monitoring data is useful for deciding whether to apply a recovery action and, if so, in identifying an appropriate time.

4.2 Error Detection

Error detection helps to apply recovery actions at an appropriate time to prevent failures from happening while minimizing user interruption. In this step, the collected data about the memory usage of an application is analyzed to detect any sign of excessive memory use. An application is said to be using memory excessively if the memory used by the application is continuously going up while the application is in the same *state*. A state is defined from the user's point of view. Two application states are the same if they look the same to

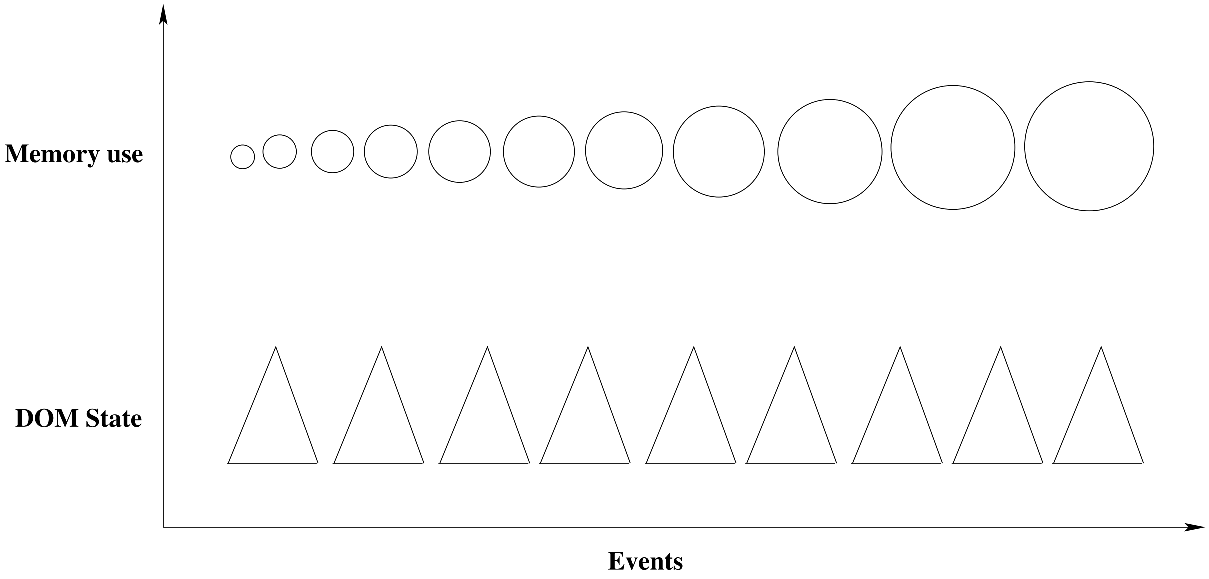


Figure 4.2: DOM state monitoring

the user. An application state is approximated by the DOM tree of the application. This approximation makes sense since the user interface of web applications is developed using the DOM. Therefore, comparing states can be done by comparing the DOM trees of an application. Continuous growth of memory usage over the same states indicates a memory leak in the system. Figure 4.2 pictures this definition of memory leak. In Figure 4.2, each triangle represents a state of the web application and each circle denotes the memory used by the application in that state (a larger circle means more memory use). MemRed does not monitor application states and assumes that the state of the application is known at each data collection point. It is a direction for future work to monitor application states. Techniques like the one proposed by Alimadadi *et al.* [2] would be useful in this regard.

To find the trend in the memory usage, the linear least squares method [101] is used to fit a line over data. The quality of the line fitted to the data is measured by computing the correlation coefficient [101] of the line. A positive slope together with a high correlation coefficient shows an increasing trend in data. The higher the correlation coefficient, the more accurate the slope of the line fitted to the data. We cannot make any conclusion about the existence of any trend in the data if the correlation coefficient is low. The slope and correlation coefficient are threshold values that should be determined based on the desired level of sensitivity.

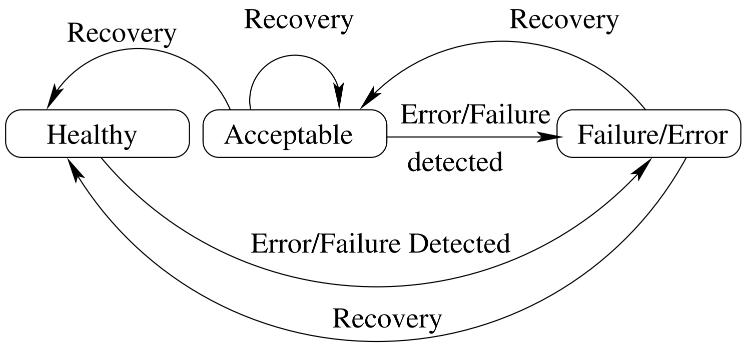


Figure 4.3: Role of recovery actions

4.3 Recovery

After an error is detected, the next step is to recover the system by removing the effects of errors. In MemRed, this is achieved by applying recovery actions on the page. Figure 4.3 shows the effects of recovery actions in different states of an application. The goal of applying the recovery actions is to return the system to a healthy state from a failure/error state, or to slow down system degradation by returning the system to an acceptable state. An acceptable state means a state where the effects of errors/failures are removed partially. The possible recovery actions in the web applications are:

- **Reloading a page**: This action loads the web application again; therefore, it cleans the internal state of the application and deletes all the leaked and unused objects.

- **Closing the page and opening it in a separate process**: This action restarts the process corresponding to the page by closing the page and opening it in a separate process. Therefore, all the components of the page are restarted. As shown in Figure 4.1, a browser tab is a separate process having three components: the JavaScript engine, which is called V8 [41] in Chrome and executes the JavaScript code of a web application; the DOM bindings, which are responsible for binding the JavaScript engine and browser code; and the HTML renderer, which displays the web page on the screen, based on the HTML code and cascading style sheets (CSS) of the web application. Closing the page restarts the states of all these components.

- **Snapshot-Reload**: It is important that the recovery actions not result in the user losing any data. In some AJAX-based web applications such as GMail, every state is represented with a URL and also benefits from an auto-saving mechanism; therefore, data loss does not happen with reloading a page; however, all applications do not benefit from auto-saving mechanisms. Therefore, we need to enrich our recovery actions to achieve recovery without data loss. To achieve this goal, the recovery actions need to be enriched by check pointing the application state. In other words, we need to save the state of a web application before performing the recovery. The state of a web application consists of a snapshot of its DOM tree as well as JavaScript functions and variables. After recovery, the application is taken to the state before the failure. We leave implementing this recovery action as future work.

- **Force Full Garbage Collection**: This action collects all the syntactic garbage on the JavaScript heap of the application. This recovery action is mentioned as an action for cleaning the internal state of an application in past studies on the server side [24]. In the context of web applications run in browsers, if a large amount of memory is not reclaimed because the browser is waiting for some threshold to perform garbage collection, the result may be excessive memory usage for some period of time. Forcing a garbage collection can help in such cases. This is true especially in the case of browsers like Chrome, where every tab is in a different process, having its own JavaScript engine. Since a full garbage collection requires traversing all the objects on the heap, we need to make sure that this action does not result in unacceptable response time.

To implement the recovery actions in the context of a browser, we can leverage its structure, which allows fine-grain control over its components; for example, in closing and opening a tab. In addition, we can take advantage of the unified user interface of web applications, which is based on the DOM.

4.3.1 Type and Time of Recovery

The decision criteria for applying recovery actions are *what* recovery actions to choose and *when* to apply a recovery action. The selection of a recovery action depends on its effectiveness in removing the error and the cost it imposes on the underlying system, in terms of user interruption and performance. With the current prototype, our goal is to study the effectiveness of the recovery actions.

The time of applying a recovery action is important. First, to prevent failures from happening the recovery actions should be applied at an appropriate time. The first indication of the time to start applying a recovery action comes from the detection phase. Second, as the systems under study, i.e., web applications, are interactive, the recovery actions should

Memory Status Tab Status	Healthy	Error	Failure
Not Visible	U	R, S, U, G, N	R, S, N
Not Focused/Visible	U	U, G, PR, PS, PN	R, S, N
Focused	U	U, G, PR, PS, PN	PR, PS, PN

R = Reload	N = Snapshot-Reload	PR = Prompt Reload
PS = Prompt Restart	S = Restart	U = Update Snapshot
G = Garbage Collection	PN = Prompt Snapshot-Reload	

Table 4.1: Acceptable recovery actions

be transparent to the user. To minimize user interruption, the recovery action should be applied at an appropriate time. Therefore, knowing whether the monitored tab is visible to the user is essential. For instance, if it is necessary to reload a page that is in the user focus, we need to wait until the page is out of the focus and then apply the recovery action. If the page does not go out of the user focus before the estimated time-to-failure, then we ask for user permission before applying a recovery action, i.e., prompt before reloading. Table 4.1 shows actions that are allowed in different scenarios considering the application's health and user visibility/focus.

4.4 Implementation

MemRed is implemented as an extension to the Chrome browser [42]. The Chrome browser has been chosen for several reasons. First, as shown in Figure 4.4, the usage share of Chrome is higher than that for all other browsers [87]. Second, the Chrome browser provides useful APIs to access its internals and develop browser extensions. In the following, we describe the implementation of different components of MemRed.

4.4.1 Monitoring and Analysis

The memory usage of an application is estimated by its heap usage. To monitor the memory usage of a web application, MemRed connects to a browser page to get data. The communication between MemRed and the browser page is done using the ChromeDevTools protocol [35], an API provided by Chrome browser to get debugging information from a page, including memory-related information such as the heap snapshot of the page.

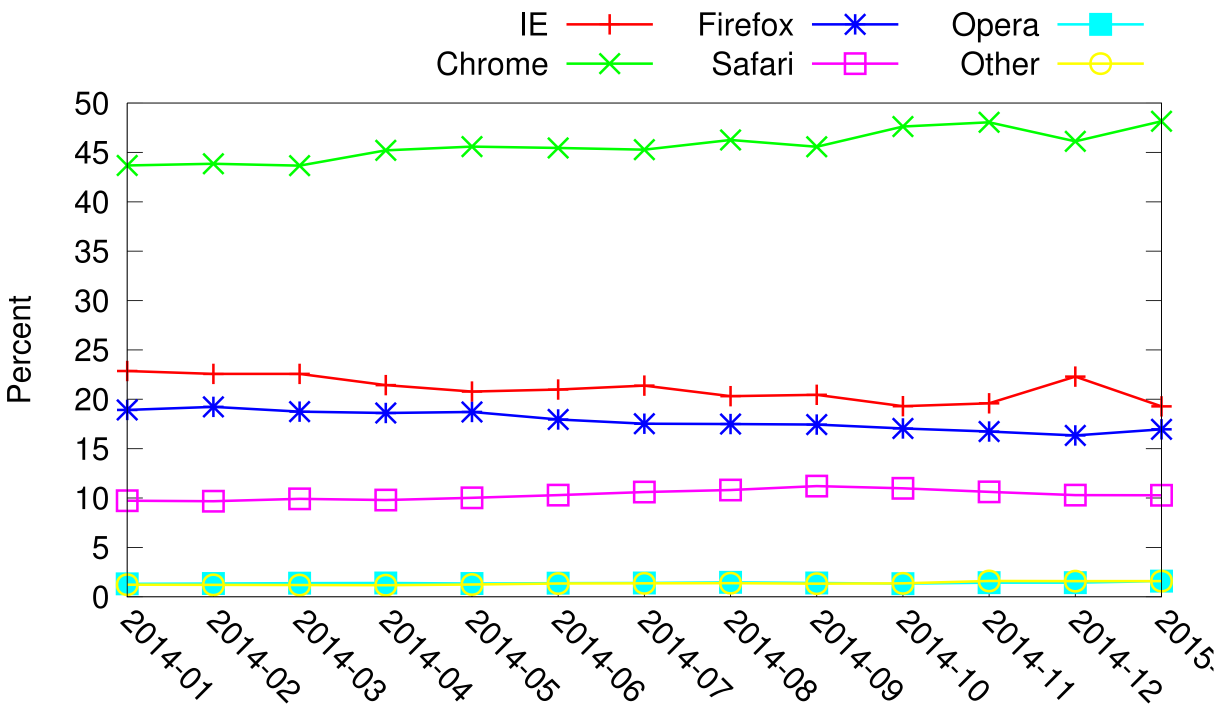


Figure 4.4: Usage share statistics of different browsers (data is taken from StatCounter [87])

MemRed collects snapshots of the JavaScript heap periodically. The heap snapshot contains the size and number of all the JavaScript objects on the heap as well as the heap structure. MemRed analyzes the collected heap snapshot to extract the metrics shown in Table 4.2. The `TotalHeapSize (JS)` metric measures the memory allocated by JavaScript engine for the heap, and `JsLive` metric denotes the size of live objects on the heap. While other metrics are useful for diagnosing the causes of the leaks, in this prototype we do not have a diagnosis step and leave that for future work. Worth mentioning is that the ChromeDevTools protocol performs full garbage collection before taking a heap snapshot. Therefore, the resulting snapshot contains all reachable objects on the JavaScript heap.

To analyze the collected data by fitting a line to the data, i.e., `TotalHeapSize` or `JSLive` over time, Weka [71] software is used. weka is an open source machine-learning library for data analysis.

4.4.2 Recovery

The recovery actions are implemented inside a browser extension by taking advantage of the Chrome extension APIs. Implementing the recovery actions inside a Chrome extension provides the opportunity to have the recovery actions at a fine granularity, compared to implementing the actions at the level of the operating system or process, which limits them to being coarse-grained actions such as rebooting a process, rebooting an operating system, or check-pointing the process.

To be able to automatically apply the recovery actions, appropriate automatic control of browser components must be available. Currently, the following recovery actions are implemented so that they can be performed programmatically:

- Reloading a page.
- Opening and closing a browser window or tab.
- Garbage collection.

In the current prototype, upon detecting an error, MemRed waits until the monitored tab is not visible to the user or the user is not actively interacting with the page, before applying the recovery actions. Garbage collection can be applied at any time.

Metric	**Description**
TotalHeapSize (JS)	The amount of memory allocated to JavaScript heap by V8 (JavaScript engine of Chrome [41]).
JSLive	Size of all live objects on the heap.
NodeCount	Total number of all objects of the application, including objects on the JavaScript heap as well as browser objects.
ArraySelf	Sum of shallow sizes of all objects that are instantiated using the array constructor function.
ArrayCount	Total number of objects that are instantiated using the array constructor.
HiddenSelf	Sum of shallow sizes of hidden objects, i.e., objects that are created behind the scene by V8.
HiddenCount	Number of hidden objects.
DocDomCount	Number of DOM objects.
DetachDomCount	Number of objects that are detached from the DOM tree.

Table 4.2: Metrics extracted from a heap snapshot

4.5 Evaluation

This section examines the effectiveness of recovery actions in removing or reducing excessive memory usage in web applications. To perform these experiments, two single-page web applications are used. The first one, Tudu [20], is an AJAX-based open-source web application used for managing personal todo lists. The client side of this application consists of 3K lines of code as well as AJAX libraries [57]. Please note that this application is different from the Todo applications used in Chapter 3 for evaluation of LeakSpot 3.

The second application under study is the GMail [37] application, which is developed mostly using JavaScript and AJAX technology. Tudu is an example of a simple and small web application into which we can simply inject memory leak bugs. On the other hand, GMail is one of the largest web applications in wide-spread use that has been developed by high-quality engineers.

In every experiment, we monitored the web application while a user navigated through the page and generated events. To expedite the effects of memory leaks, we simulated user navigation using the WebDriver API [84] for Java. To alleviate the need for application-state monitoring, as discussed in Section 4.2, we forced the simulated user to do a set of specific tasks repetitively; therefore, the application state remains unchanged. An increasing trend in memory usage over time for such a repetitive scenario indicates excessive memory usage in the application, since it implies that all the memory allocated is not released.

Scenario	Description
TuduNoAction	Simulated user is navigating through the Tudu page and doing the set of actions repeatedly.
TuduReload	Simulated user is navigating through the Tudu page similarly to the TuduNoAction case; the page is reloaded periodically.
TuduDiff	Simulated user is navigating through the Tudu page similarly to the TuduNoAction case; the page is closed and opened in a new tab (actually in a new process) periodically.

Table 4.3: Different cases of data collection on the Tudu application

The experiments are performed on a 2.4 GHz quad-core PC with $4GB$ of RAM. In the following, the case in which a leak is injected into the Tudu application is presented, first. Then, the results of experiments on the GMail application are shown.

4.5.1 Tudu Experiment

This section presents the results of experiments studying the effectiveness of recovery actions on Tudu. To do the experiments, a memory leak is injected into the Tudu application. The memory leak is so that every time the user adds a list, some objects are created on the JavaScript heap but are not used or deleted which results in a memory leak in the application. As defined earlier in Chapter 2, a memory leak means the existence of unused objects, i.e., objects that are accessible from the garbage-collection roots but are never used by the application [76]. To expedite the effects of a memory leak, after logging into the Tudu application the simulated user does the following actions iteratively while waiting four seconds after every action:

1. Add a new todo list.
2. Edit the list.
3. Delete the list.

These actions are performed so that at all times, there are at most two lists in the system. To see the effects of recovery actions, data is collected while the Tudu application is run under different scenarios as specified in Table 4.3. The TuduReload case, in which the page is reloaded periodically, shows the effectiveness of reloading a page. The TuduDiff case, in which the browser tab executing the application is periodically closed and opened, shows the effectiveness of opening/closing a tab. The TuduNoAction case provides a baseline for comparison with the cases where recovery actions are applied. In every test case, monitoring data is collected every minute. Note that in the interactive web applications under study, the memory leak is a function of the number of events occurring on the page; however, since the user is repeating the same set of tasks and the time interval between repetitions is equal, time is a good approximation of the number of events occurring on the page.

Figure 4.5 shows the size of live objects on the heap (`JSLive`) for different scenarios on the Tudu application. As shown in the figure, there is an increasing trend in size of objects on the heap in the TuduNoAction. This increase occurs because there is a memory leak in the application but no recovery action is applied. As shown in Figure 4.5, there is

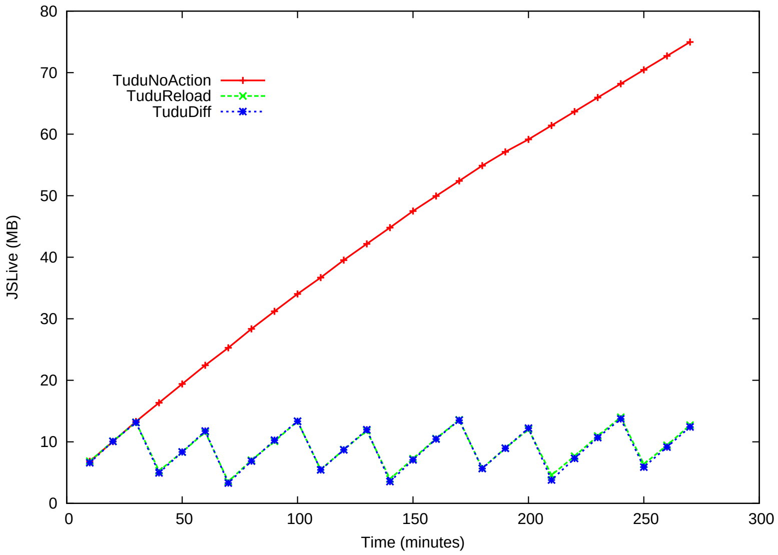


Figure 4.5: JSLive over time for different test cases on the Tudu application

no increasing trend in the cases where recovery actions are applied. The oscillation in data values is due to performing a recovery action periodically.

To measure the trend in data mathematically, a line is also fitted to the collected data to compute any trend in this data. The slopes and correlation coefficients of the corresponding lines are shown in Table 4.5. A positive slope accompanied by a high correlation coefficient (larger than 0.7) indicates an increasing trend. Please note that the graph in Figure 4.5 is similar to the time-count graph in Chapter 3, in which a positive slope indicates a memory leak, unlike the collective-staleness graph in which a diagonal line indicates a memory leak. To compute the slope for the TuduReload and TuduDiff test cases, the data is smoothed using ten neighboring points to remove the effect of data oscillation on trend estimation. As can be seen, there is no trend in the data when recovery actions are applied periodically. These results support the idea that excessive memory usage by a web application can be mitigated by applying recovery actions.

4.5.2 GMail Experiments

Scenario	Description
GmailNoAction	Simulated user is navigating through the page and doing the set of actions repeatedly; no recovery action is applied.
GmailReload	Simulated user is navigating through the page as in the GmailNoAction case; the page is reloaded periodically.
GmailDiff	Simulated user is navigating through the page as in the GmailNoAction case; the tab is closed and opened in a new tab (actually in a new process) periodically.
GmailIdle	The Gmail page is open and being monitored but the simulated user does not do any action.
GmailIdleFullGC	Gmail page is open and being monitored; the simulated user does not do any action; periodically a full garbage collection is performed.

Table 4.4: Different cases of data collection on the GMail application

This section discusses the result of experiment on GMail. The simulated user, after logging into her GMail account, performs the following actions iteratively while pausing a few seconds after every action:

1. Composes an email (she just clicks on the compose button without sending any email to prevent generating new objects).
2. Clicks on the list of important emails.
3. Clicks on the Inbox button.
4. Reads an email (the same email is read in each iteration to avoid new object generation).

Different test cases on GMail application are shown in Table 4.4. As can be seen in Figure 4.6, in the case of the GmailNoAction where no recovery action is applied on the

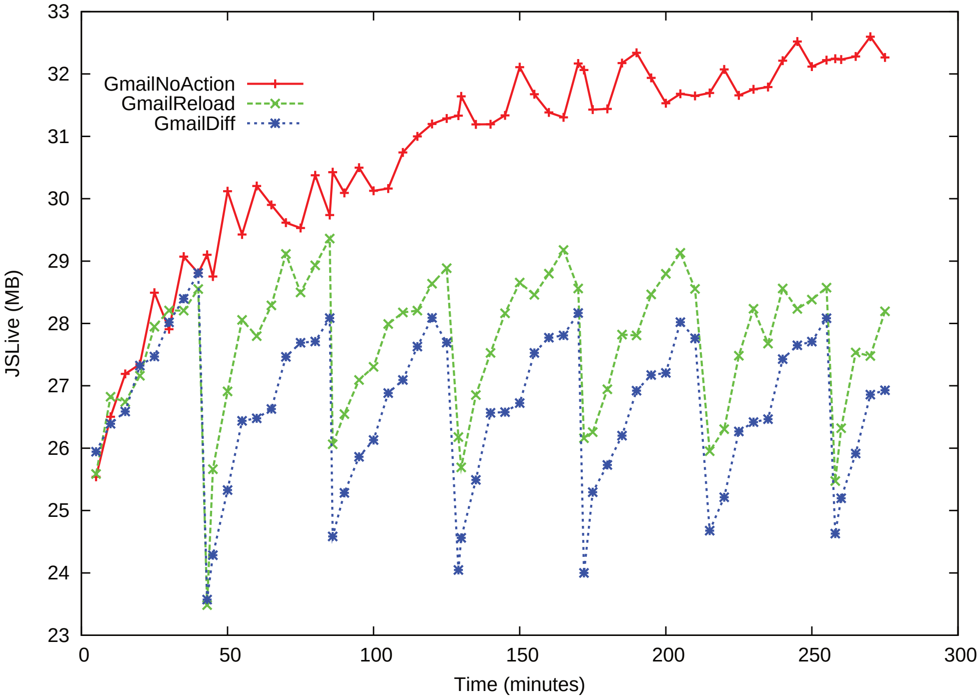


Figure 4.6: JSLive over time for different test cases on the GMail application

page, the total size of objects on the heap increases. In this test case, the simulated user is doing a repeated set of tasks without generating new objects (it is not sending or receiving any email). Therefore, any growth in the heap size indicates the existence of memory leaks in the GMail application. The existence of memory leaks in GMail has been discussed by the developers of Google [56]. Note that since the ChromeDevTools protocol performs a full garbage collection before taking a heap snapshot, this growth cannot be related to a delay in garbage collection. The important point is that this application suffers from excessive memory usage. It is possible that the application is intended to function in this way. For example, the application may keep a history of all the states; the user may not refer to them in the future, which results in a growth in the number and size of the objects on the heap. This behaviour is not acceptable in an operational context since it results in unbounded memory growth, which affects the reliability and response time of the system. Therefore, having a mechanism to reduce excessive memory usage is crucial.

Figure 4.6 also shows the result of the GmailReload test case, where we periodically reload the page, and the GmailDiff test case, where we periodically close and open the page. We do not see an increasing trend when recovery actions are applied, since the recovery

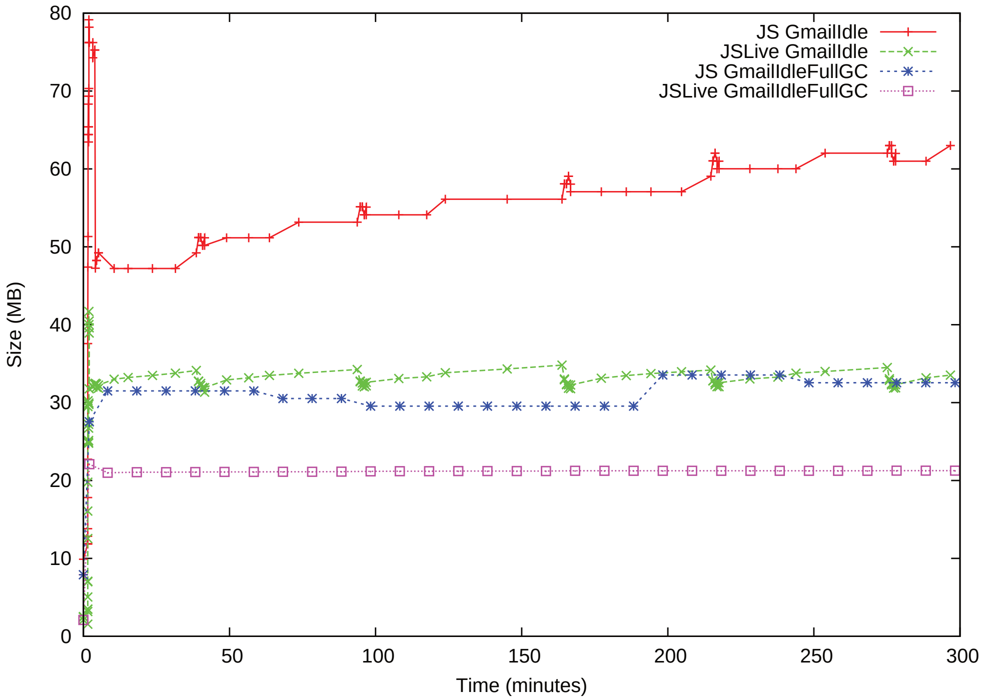


Figure 4.7: JSLive and JS over time for different test cases on the GMail application

	JSLive	
	Slope	**CC**
TuduNoAction	0.26	0.9976
TuduReload	0	0.1283
TuduDiff	0	0.1267
GmailNoAction	0.017	0.7011
GmailReload	0	0.306
GmailDiff	0	0.0807
GmailIdle	0	0.1019

Table 4.5: Slope and correlation coefficient (CC) of JSLive for different cases

actions delete leaked objects from the heap. The results shown in Table 4.5 also confirm this. Table 4.5 shows the slopes and correlation coefficients of the lines fitted to the data;

the unit of slope is MB/minute. To compute the parameters of this line, we smoothed each data point using ten neighboring points. The reason for this smoothing is that the data values for GmailReload and GmailDiff are oscillating between two boundaries because of periodic recovery actions. As can be seen, there is no increasing trend when the recovery actions are applied.

To measure the effectiveness of the garbage collection action, we need to look at the memory allocated by V8 for the heap, which is measured by the TotalHeapSize (JS) metric, as shown in Table 4.2. To study the garbage collection action, we perform two experiments, with the user idle in both cases. In GmailIdleFullGC, we periodically collect all garbage and in GmailIdle, we apply no recovery action. Figure 4.7 shows the values of the `JSLive` and `JS` metrics over time. As can be seen, the memory that is allocated by the V8 JavaScript engine for the heap, i.e., JS, is going up for the GmailIdle experiment; however, we do not see such a trend when garbage is collected periodically. Also, there is no increasing trend in the size of objects on the heap (JSLive). Note that even though the simulated user is idle, there are some events fired on the page using a timer.

The increasing trends observed in metrics collected from the GMail application represent memory leaks in a commonly used web application where the user is doing a set of simple tasks. In a real world scenario in a browser, we are dealing with large numbers of tabs, each one executing a different web application. We also need to consider that GMail is a well-engineered application, which is not the case for all those applications running in different tabs. Therefore, we need a mechanism for improving the reliability of the client side of web applications.

4.6 Conclusion

In this chapter, we presented our approaches for managing memory leaks at runtime on the client side of web applications. We presented the overall solution and the architecture of MemRed, a prototype tool for removing/reducing leaked objects at runtime. MemRed, which is implemented as an extension to the Chrome browser, monitors the memory usage of a web application by periodically collecting heap snapshots. It then analyzes the collected data to detect trends in memory usage of the corresponding web application. Upon detecting a possible memory leak, it applies a recovery action to remove the effects of failures or slow down system degradation. It tries to apply recovery actions at an appropriate time, so that the impact on the user is minimal. Our evaluation demonstrates the existence of memory leaks in a complex and popular web application and the effectiveness of recovery actions in removing/reducing the effects of the leaks.

Chapter 5

Conclusion and Future Work

This chapter summarizes the contributions of the thesis in Section 5.1 and discusses directions for future work in Section 5.2.

5.1 Summary

Memory leaks, the existence of unused objects on the heap of an application, result in performance degradation and even crashing of the application. With the rise of single-page applications, long-running JavaScript is becoming more prevalent and with it the potential for memory leaks. This thesis has studied the extent of memory leaks in JavaScript applications and proposed two different approaches for tackling this problem. First, it introduced LeakSpot, a tool for detection and diagnosis of memory leaks during the development of applications. LeakSpot is shown to be effective in finding and fixing memory leaks in complex benchmarks and open source web applications. In addition, LeakSpot is shown to be scalable by applying it to real and complex web applications such as GMail. The effectiveness of LeakSpot in highlighting the potential causes of leaks in web applications has also been demonstrated experimentally. Second, it proposed MemRed as a tool to improve the memory efficiency of web applications. MemRed works by detecting excessive memory use in web applications and applying recovery actions to increase memory efficiency and reduce the effects of memory leaks. MemRed is shown to be effective in reducing memory usage of benchmarks and open-source web applications.

5.2 Future Work

This section discusses different directions for future work.

5.2.1 Runtime Management of Memory Leaks: A Different Approach

To manage memory leaks at runtime, this thesis proposed MemRed to remove the leaked objects. As another approach, the leaked objects could be *tolerated* by moving them to disk. The idea of tolerating memory leaks has already been used in other programming languages. Plug [67] has been proposed for C/C++ applications and LeakSurvivor [90] and CRAMM [108] have been proposed for garbage collected languages; however, these tools cannot be used with JavaScript, e.g., in browsers.

Implementing this approach requires two main components. First, it needs to determine which objects should be moved to disk. To select these objects, the techniques developed in LeakSpot can be used. Second, it requires a mechanism to move objects to disk and a reclamation policy to get those objects from the disk, if needed, without affecting program execution. Such a policy should take care of all the references to the objects.

This new approach of moving the unused objects to the disk could improve the performance of the memory management system. First, as studied by Aigner *et al.* [1], the performance of the memory system, e.g., the allocation time of objects, depends on the size and liveness of objects on the heap. Therefore, moving long-lived unused objects to disk can improve the performance of memory management systems. Second, moving objects to disk means lighter workload for garbage collection, which results in shorter pause times and less frequent garbage collection.

5.2.2 Online Use of LeakSpot

Although LeakSpot helps developers in finding and fixing leaks during development. The design decisions in LeakSpot make it possible to use it in an online setting. First, the source-level instrumentation and use of a proxy make LeakSpot portable across different browsers. In addition, monitoring runtime behaviour of objects by modifying the source code of the applications makes it possible to apply optimizations to improve performance without decreasing leak-detection accuracy. In future work, we intend to apply the discussed optimizations in Section 3.6.5 to reduce LeakSpot overhead.

5.2.3 Memory Leak Detection Based on State Monitoring

In MemRed, to detect memory leaks at runtime, we used heap growth as an indication of memory leaks. To make it more precise, we could combine state monitoring with heap growth to detect memory leaks in a page. To achieve this goal, we need to monitor the DOM state of a page to find any changes in that state.

5.2.4 Alternative Heuristics for Memory Leak Detection

One direction for future work is to apply different heuristics or improve the current heuristics to find the causes of memory leaks in LeakSpot. First, the profiling data could be used to find the causes of memory leaks by tracking the number of references created to or removed from the objects, similarly to what is done in LeakPoint [13] for detecting leaked objects. Implementing this approach requires monitoring reference removals. Second, the allocation sites and reference sites heuristics in LeakSpot could be improved by categorizing objects based on different criteria such as the number or type of references to the objects. Third, another heuristic would involve monitoring the reference creation and removal history for each object, including the time of reference creation/removal and the type of references. The idea behind this approach is that all the references to objects that are temporally close to each other need to be added or removed in the same time frame. This technique would be especially useful for finding closure memory leaks. This kind of memory leak happens when some of the references are gone (local references) while other references (references from the enclosing functions) keep the closure variables alive. Finally, while LeakSpot focuses on finding the problematic locations in the program, the object-level data could be used to find problematic data structures by studying and finding the patterns of reference creation or removal to objects. Achieving this goal requires monitoring reference removals in addition to the reference creation monitoring currently supported in LeakSpot.

5.2.5 Enriching Recovery Actions

In the current prototype, MemRed applies recovery actions to remove the effects of leaked objects; however, some of the actions, such as reloading a page, may cause the data of a page to be lost. To achieve recovery without data loss, the recovery actions need to be enriched by check pointing the application state. The applications used for evaluating MemRed, such as GMail and Tudu, are designed so that they preserve the state of applications at every moment.

5.2.6 Applicability to Other Domains

This thesis has focused on client-side web development, even though memory leaks are still a bigger problem for server-side JavaScript [28, 44], e.g., applications running on Node.js [43], as the server-side programs often implement long-running applications. The techniques used for memory leak detection on the server side either focus on performance profiling [5] or tools to navigate through the heap [28] that are not sufficient for finding the causes of the leaks. The techniques used in LeakSpot can be used for JavaScript on the server side, since we are not limited by computational resources on the server side.

APPENDICES

Appendix A

Implementation of Proxy

This appendix first explains the architecture of proxy used in LeakSpot. Then, it describes how the proxy handles SSL-related errors.

A.1 Proxy Architecture

To modify the JavaScript code of a page, the proxy intercepts all the messages between the client and server. To make it possible for the proxy to work for all different web pages, it handles both HTTP and HTTPS connections:

- HTTP connections: In the case of an HTTP connection, the proxy simply forwards the request from the client to the server, modifies the replies from the server, and forwards the modified replies to the client.

- HTTPS connections: The HTTPS proxy just relays the data without knowing anything about its content. In this case, to be able to intercept data, the MITM proxy sits in the connection between the client and the server. It replies to the initial request by the client, so the client assumes it is the server, while acting as the client for the server. As shown in Figure A.1, the HTTPS proxy forwards data between the client and MITMP proxy without knowing anything about the content of the data by creating a secure channel (pipe). It cannot view or manipulate the data since the data is encrypted. Please note that such a secure channel is not needed for HTTP connections.

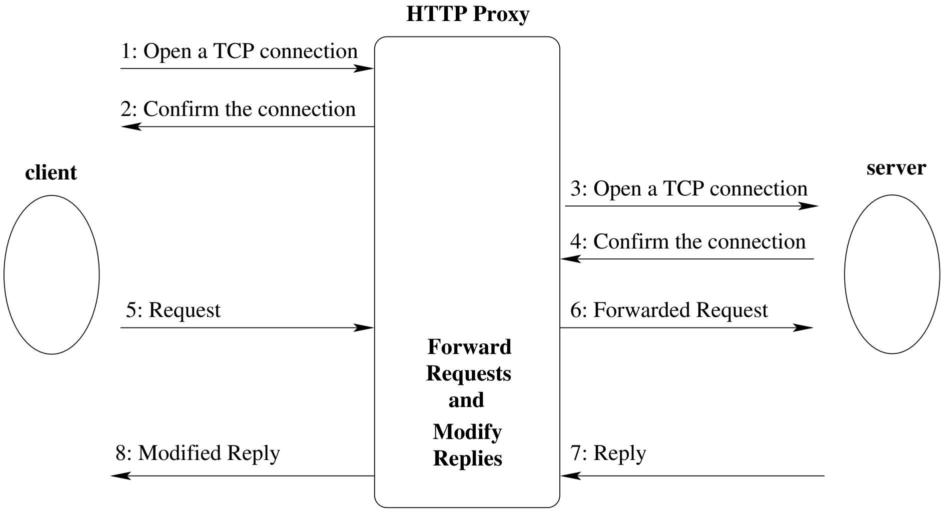


Figure A.1: Handling HTTP connections by the proxy.

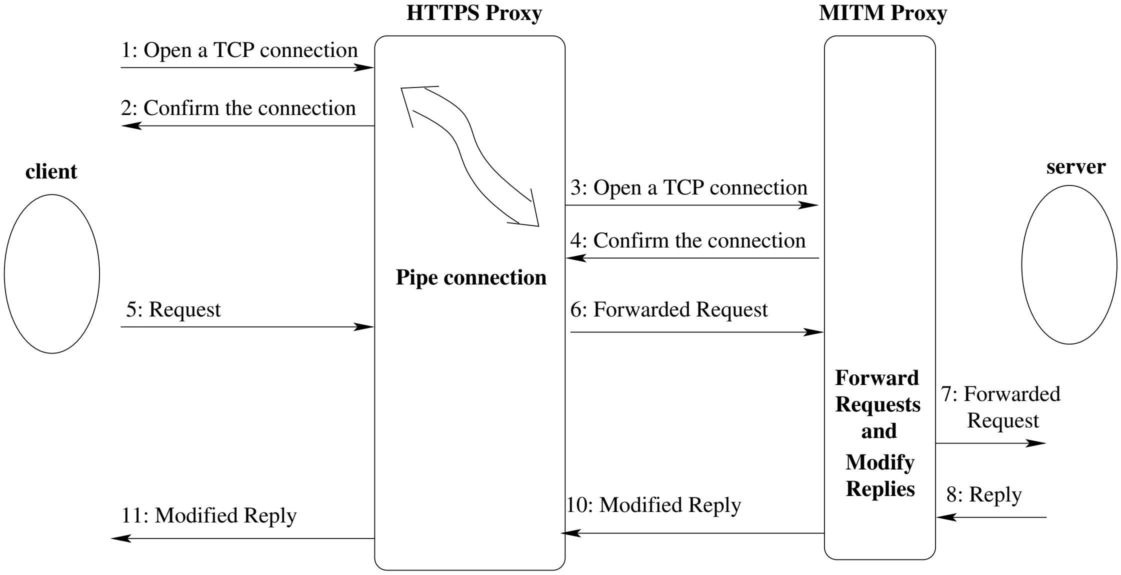


Figure A.2: Handling HTTPS connections by the proxy

A.2 Dealing with SSL-Related Errors

Whenever there is a man-in-the-middle attack, the SSL protocol detects and lets the browser know that there is a problem and, so, it gives a warning. It is possible to develop the proxy in a way so that the browser is not notified, as discussed by Marlinspike [55]; however, going into these details is beyond the scope of this work. To avoid the warnings generated because of man-in-the-middle attacks, the different warning messages are handled as follows.

- **Certificate authority is not trusted by the browser**: The certificate used by the MITM proxy, when acting as a server, is self-signed, so it is not trusted by the client, i.e., browser. The browser allows the communication to proceed by giving the user a notification. To avoid this warning, the certificate authority needs to be known to the browser.

- **" This is not probably the site you are looking for"**: This warning is generated when the domain name in the common name field of the certificate differs from the domain requested by the client. This error is especially problematic when the client needs to connect to a page containing several domains. When connecting to a single site through the browser, it is possible to manually accept the SSL certificate and move forward; however, in cases where it is necessary to connect to different domains to fully load a page, using a single-domain certificate results in the page not loading fully. To avoid this problem, multi-domain certificats are used. Creating multi-domain certificates for a page requires knowledge of the different domains in the page. This data can be extracted by accessing the page through the proxy, and recording all the different domains. This process is done just once for every page. Implementing the proxy so that the certificates are generated on the fly for every domain could be a subject of future work.

Appendix B

DOM Objects

LeakSpot works by monitoring the allocations, accesses, and reference creations of JavaScript objects. The heap of a web application is composed of JavaScript and DOM objects. DOM objects behave differently from ordinary JavaSceript objects in terms of allocations, accesses, and reference creations. For example, a new `div` element is created with a call to `document.createElement('div')`, not with `new HTMLDivElement`, although all `div` elements inherit from the `HTMLDivElement.prototype` [48]. In addition, references are created between objects by calling specific functions such as `element.AppendChild` which creates a reference between two DOM objects. Therefore, to monitor activities on different types of objects, all the different ways that objects are allocated, accessed, or referenced should be considered. In addition, managing the memory of DOM objects is done differently in various environments. This appendix explains why allocating and releasing the memory of DOM objects are done differently than is the case for JavaScript objects. Then, it lists some of the DOM APIs that are used for allocations, accesses, and creating references on DOM objects.

B.1 DOM Object Allocations and Memory Management

This section discusses why DOM object creations are different from JavaScript object creations and why DOM does not address memory management. DOM APIs are designed to be compatible with a wide range of languages; they are interfaces rather than classes. Thus, an implementation of DOM APIs needs only to expose methods with the defined names and specified operation, not implement classes that correspond directly to the interfaces. Therefore, DOM APIs can be implemented in a wide variety of environments and applications where scripts within an HTML document are one case. Since DOM APIs are interfaces, ordinary object constructors (in the Java or C++ sense) cannot be used to create DOM objects. Therefore, factory methods are defined to create instances of objects that implement the various interfaces. Objects implementing some interface `X` are created by a `createX()` method on the Document interface since all DOM objects live in the context of a specific Document.

Moreover, DOM does not handle memory management issues at all. The Core DOM APIs need to operate across a variety of memory management systems, from those (notably Java) that provide explicit constructors but provide an automatic garbage collection mechanism to automatically reclaim unused memory, to those (especially C/C++) that generally require the programmer to explicitly allocate object memory, track where it is used, and explicitly free it for re-use. To ensure a consistent API across these platforms, the DOM does not address memory management issues but instead leaves these for the implementation.

B.2 DOM APIs

This section provides an example list of the interfaces and their functions and properties that need to be intercepted in order to monitor DOM allocations, accesses and reference creations. In the listings presented here, only properties that may be modified by the application are listed, since some properties may be read-only. In addition, the complete wrapping of all the functions, which is a significant engineering effort, could be tackled as a community effort. The full list of methods and properties is available [99]. The DOM presents documents as a hierarchy of Node objects that also implement other, more specialized interfaces such as Element, Node, and HTMLElement [97].

- Document: The Document interface represents the entire HTML document. Conceptually, it is the root of the document tree, and provides the primary access to the document's data. Since elements, text nodes, comments, processing instructions, etc., cannot exist outside the context of a Document, the Document interface also contains the factory methods needed to create these objects. Table B.1 lists some of the methods and properties that allocate an object of this type or create a reference to an object of this type.

- Node: Every DOM node object should implement the `Node` interface. Table B.3 lists some of the methods and properties that allocate, access, or create references to these types of objects.
- Element: Table B.4 lists some of the methods and properties that allocate, access or create references to objects of this type. For example, DOM objects could be created by setting the `innerHTML` properties of DOM elements to a string, such as:

  `element.innerHTML = "<iframe id='testframe'>"`;

  To monitor this method of object allocation, we need to add more functionality to the `$.logPutFieldDot` logging function to provide different handling for cases in which the name of the property is `innerHTML`.
- HTMLDocument: All Document objects must also implement the HTMLDocument interface. The HTMLDocument interface has several functions that create or remove DOM objects, such as `write, open,` and `writeln`. Table B.2 lists some of the methods and properties that allocate, access, or create references to these objects.

The functions and properties of the NodeList interface need to be considered as well. This interface handles ordered lists of Nodes, such as the children of a Node or the elements returned by functions such as `getElementsByTagName`, which is a method of the Element interface. As well, the functions of the HTMLCollection interface should be cosidered, since the output of many functions is an HTMLCollection.

The event handlers that are bound to a DOM element also need to be considered, since they create references between the DOM element and the event handlers. There are three methods for registering event handlers for DOM elements in JavaScript [2], depending on which DOM Level[1] they are using. The event handler registration in DOM Level 2 is done programmatically using the addEventListener function, i.e., `e.addEventListener` methods [98] in the JavaScript code. In DOM Level 1, the event handlers are bound to a DOM element by assigning the attributes of an `HTMLElement`, e.g, `e.onclick=function(event){}`. In both cases, to monitor the references created between objects, the corresponding functions are wrapped so that the DOM element, the event listener attached, and the reference created between them are logged during runtime. In DOM Level 0, event handlers are inlined in the HTML code, for example:

[1] DOM Levels are the versions of the specification for defining how the DOM should work. Each new level of the DOM adds or changes specific sets of features. DOM Level 0 supports accesses to a few HTML elements, most importantly forms and images. DOM Level 1 defines the core elements of the Document Object Model. DOM Level 2 extends those elements and adds events. DOM Level 3 extends DOM level 2 and adds more elements and events.

	Type	Name	Return Value
Object Allocation	Methods	createElement createDocumentFragment createTextNode createComment createAttribute createElementNS createAttributeNS createRange importNode	Element DocumentFragment Text Comment Attr Element Attr Range Node
	Properties	innerHTML	–
Object Accesses	Methods	getElementsByTagName getElementById getElementsByTagNameNS querySelector querySelectorAll elementFromPoint(Number x, Number y)	NodeList Element NodeList Element Element Element
	Properties	documentElement childNodes	Element NodeList
Reference creation	Methods	–	–

Table B.1: Document object interface

`<button onclick="handler()">.`

Monitoring the object and the reference created using this method can be done using the techniques proposed by Alimadadi *et al.* [2], i.e., removing the inline-registered listener from its associated HTML element, annotating the HTML elements, and re-registering the listener using the `addEventListener` function. In this way, the event handlers can be handled similarly to the previous two cases.

	Type	Name	Return Value
Object Allocation	Methods	open open close, write, writeln	HTMLDocument window void
Object Accesses	Properties	body images embeds plugings links forms anchors scripts childNodes activeElement	HTMLElement HTMLCollection HTMLCollection HTMLCollection HTMLCollection HTMLCollection HTMLCollection HTMLCollection NodeList object Element
	Methods	commands getElementsByName getElementsByClassName	HTMLCollection NodeList NodeList
Reference creation	Methods	–	–

Table B.2: HTMLDocument object interface

	Type	Name	Return Value
Object Allocation	Methods	cloneNode	–
Object Accesses	Properties	nodename nodeValue nodeType parentNode childNodes firstChild lastChild previousSibling nextSibling attributes childNodes ownerDocument	Node Node Node Node Node Node Node Node Node Node NodeList Document
	Methods	normalize	–
Reference creation	Methods	appendChild replaceChild insertBefore	Node Node Node

Table B.3: Node object interface

	Type	Name	Return Value
Object Allocation	Properties	innerHTML outerHTML	– –
Object Accesses	Properties	getElementsByTagName getElementsByTagNameNS getElementsByClassName querySelector querySelectorAll	NodeList NodeList HTMLCollection Element NodeList
Reference creation	Methods	insertAdjacentHTML	void

Table B.4: Element object interface

References

[1] Martin Aigner, Thomas Hütter, Christoph M Kirsch, Alexander Miller, Hannes Payer, and Mario Preishuber. ACDC-JS: Explorative benchmarking of JavaScript memory management. In *Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 67–78. ACM, 2014.

[2] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 367–377. ACM, 2014.

[3] Javier Alonso, Jordi Torres, Josep Lluis Berral, and Ricard Gavalda. Adaptive online software aging prediction based on machine learning. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 507–516. IEEE, 2010.

[4] Silviu Andrica and George Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410. IEEE, 2011.

[5] AppDynamics. Performance analytics for node.js. http://nodetime.com/. [Accessed March 2015].

[6] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. A planning based approach to failure recovery in distributed systems. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems*, pages 8–12. ACM, 2004.

[7] Algirdas Avizienis, J.-C. Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[8] Michael D Bond and Kathryn S. McKinley. Bell: Bit-encoding online memory leak detection. *ACM Sigplan Notices*, 41(11):61–72, 2006.

[9] Almende B.V. Dynamic, browser-based visualization library. https://github.com/almende/vis. [Accessed March 2015].

[10] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot-a technique for cheap recovery. In *OSDI*, volume 4, pages 31–44, 2004.

[11] Vittorio Castelli, Richard E. Harper, Philip Heidelberger, Steven W. Hunter, Kishor S. Trivedi, Kalyanaraman Vaidyanathan, and William P. Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, 2001.

[12] Chromium Issues. High memory usage when periodically loading a new document in an iframe. https://code.google.com/p/chromium/issues/detail?id=359401. [Accessed August 2014].

[13] James Clause and Alessandro Orso. LEAKPOINT: Pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 515–524. ACM, 2010.

[14] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. Software aging analysis of the Linux operating system. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 71–80. IEEE, 2010.

[15] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[16] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[17] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the specjvm98 java benchmarks. In *ECOOP99 Object-Oriented Programming*, pages 92–115. Springer, 1999.

[18] The Dojo Foundation. Dojo toolkit. http://dojotoolkit.org/. [Accessed April 2014].

[19] Ben Dolmar. Understand memory leaks in JavaScript applications. http://www.ibm.com/developerworks/library/wa-jsmemory/index.html?utm_source=tuicool. [Accessed April 2014].

[20] Julien Dubois. Tudu lists. http://www.julien-dubois.com/tudu-lists. [Accessed March 2015].

[21] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 59–70. ACM, 2008.

[22] ECMA International. Ecmascript language specification, 3rd edition, ECMA-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm. [Accessed April 2014].

[23] Asger Feldthaus, Max Schafer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[24] Sachin Garg, Aad van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. A methodology for detection and estimation of software aging. In *The Ninth International Symposium on Software Reliability Engineering*, pages 283–292. IEEE, 1998.

[25] David Glasser. A surprising JavaScript memory leak found at meteor. http://point.davidglasser.net/2013/06/27/surprising-javascript-memory-leak.html. [Accessed August 2014].

[26] Michael Grottke, Lei Li, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. Analysis of software aging in a web server. *Reliability, IEEE Transactions on*, 55(3):411–420, 2006.

[27] Salvatore Guarnieri and V. Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, volume 10, pages 78–85, 2009.

[28] Mozilla Hacks. Tracking down memory leaks in node.js a node.js holiday season. https://hacks.mozilla.org/2012/11/

tracking-down-memory-leaks-in-node-js-a-node-js-holiday-season/. [Accessed Febuary 2015].

[29] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *ACM SIGPLAN Notices*, 39(11):156–164, 2004.

[30] Ariya Hidayat. Esprima. https://github.com/ariya/esprima. [Accessed April 2014].

[31] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 381–390. IEEE, 1995.

[32] Alexa Internet Inc. Alexa: The top 500 sites on the web. http://www.alexa.com/topsites. [Accessed March 2014].

[33] Google Inc. AngularJS: HTML enhanced for web apps. https://angularjs.org/.

[34] Google Inc. Chrome developer tools. https://developers.google.com/chrome-developer-tools/docs/overview. [Accessed May 2012].

[35] Google Inc. Chrome Devopers Tools protocol. http://code.google.com/p/chromedevtools/wiki/ChromeDevToolsProtocol. [Accessed May 2012].

[36] Google Inc. Closure library. https://developers.google.com/closure/library/. [Accessed May 2013].

[37] Google Inc. GMail: A Google approach to email. http://gmail.com. [Accessed May 2012].

[38] Google Inc. Google docs. https://docs.google.com/. [Accessed May 2012].

[39] Google Inc. LeakFinder for JavaScript. http://code.google.com/p/leak-finder-for-javascript/. [Accessed Febuary 2013].

[40] Google Inc. V8 Benchmarks. http://v8.googlecode.com/svn/data/benchmarks/current/run.html. [Accessed August 2014].

[41] Google Inc. V8 JavaScript engine. http://code.google.com/apis/v8/design.html. [Accessed May 2012].

[42] Google Inc. Chrome browser. *https://www.google.com/chrome*, 2012. [Accessed May 2012].

[43] Joyent Inc. Nodejs. http://nodejs.org. [Accessed Nov. 2012].

[44] Joyent Inc. Walmart node.js memory leak. https://www.joyent.com/developers/videos/walmart-node-js-memory-leak-part-1. [Accessed March 2015].

[45] Yahoo Inc. Yahoo JavaScript library (YUI). http://yuilibrary.com/. [Accessed April 2014].

[46] Michael Isard. Autopilot: Automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[47] V8 JavaScript Engine Issues. Memory leak caused by inline caches. https://code.google.com/p/v8/issues/detail?id=2683. [Accessed May 2014].

[48] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 59–69. ACM, 2011.

[49] JQuery Foundation. JQuery. http://jquery.com/. [Accessed April 2014].

[50] Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. *ACM SIGPLAN Notices*, 42(1):31–38, 2007.

[51] Emre Kiciman and Benjamin Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. *ACM SIGOPS Operating Systems Review*, 41(6):17–30, 2007.

[52] Erick Lavoie, Bruno Dufour, and Marc Feeley. Portable and efficient run-time monitoring of JavaScript applications using virtual machine layering. In *ECOOP 2014–Object-Oriented Programming*, pages 541–566. Springer, 2014.

[53] Benjamin Livshits and Emre Kiciman. Doloto: Code splitting for network-bound Web 2.0 applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 350–360. ACM, 2008.

[54] Gen Lu and Saumya Debray. Automatic simplification of obfuscated JavaScript code: A semantics-based approach. In *IEEE Sixth International Conference on Software Security and Reliability (SERE)*, pages 31–40. IEEE, 2012.

[55] Moxie Marlinspike. New tricks for defeating SSL in practice. http://www.thoughtcrime.org/software/sslstrip/. [Accessed March 2015].

[56] John McCutchan and Loreena Lee. Effectively managing memory at GMail scale. http://www.html5rocks.com/en/tutorials/memory/effectivemanagement/. [Accessed April 2014].

[57] Ali Mesbah. *Analysis and testing of AJAX-based single-page web applications*. PhD thesis, Delft University of Technology: TU Delft, 2009.

[58] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *NSDI*, volume 10, pages 159–174, 2010.

[59] Michael S. Mikowski and Josh C. Powell. *Single Page Web Applications*. Manning Publications, 2013.

[60] Nick Mitchell and Gary Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP 2003–Object-Oriented Programming*, pages 351–377. Springer, 2003.

[61] Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. *ACM SIGPLAN Notices*, 42(10):245–260, 2007.

[62] Mozilla. LeakGauge. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Leak_Gauge. [Accessed March 2015].

[63] Mozilla. Performance. https://developer.mozilla.org/en-US/docs/Mozilla/Performance. Accessed Aug. 2014.

[64] Mozilla Developer Network and individual contributors. Gecko DOM reference. https://developer.mozilla.org/en/Gecko_DOM_Reference. Accessed 2013.

[65] Mozilla Developer Network and individual contributors. Navigation timing API. https://developer.mozilla.org/en-US/docs/Navigation_timing. [Accessed April 2014].

[66] Mozilla Developer Network and individual contributors. Parser API. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API. [Accessed June 2014].

[67] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Plug: automatically tolerating memory leaks in C and C++ applications. *University of Massachusetts*, 2008.

[68] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. *ACM Sigplan Notices*, 44(6):397–407, 2009.

[69] F.S. Ocariza Jr, K. Pattabiraman, and B. Zorn. Javascript errors in the wild: An empirical study. In *Proceedings of 22nd International Symposium on Software Reliability Engineering*, 2011.

[70] Octane. Octane Benchmark, 2014. Accessed Aug. 2014.

[71] The University of Waikato. Weka–data mining with open source machine learning software. [Accessed May 2012].

[72] Stack Overflow. Memoy leak due to JavaScript closure example. http://stackoverflow.com/questions/19798803/how-javascript-closures-are-garbage-collected. Accessed May 2014.

[73] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.

[74] Jacques A. Pienaar and Robert Hundt. JSWhiz: Static analysis for JavaScript memory leaks. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–11. IEEE, 2013.

[75] Bruce Ratner. The correlation coefficient: Definition. http://www.dmstat1.com/res/TheCorrelationCoefficientDefined.html. [Accessed March 2015].

[76] Derek Rayside and Lucy Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, pages 194–203. ACM, 2007.

[77] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232. ACM, 2009.

[78] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. *ACM SIGPLAN Notices*, 46(10):677–694, 2011.

[79] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. *ACM Sigplan Notices*, 45(6):1–12, 2010.

[80] Tom Robinson. Automagically wrapping JavaScript callback functions. http://tlrobinson.net/blog/2008/10/wrapping-javascript-callbacks/. Accessed 2013.

[81] Masoomeh Rudafshani and Paul AS Ward. Towards dependable clients: Improving the reliability and availability of the browsers. In *Proceedings of the 9th Middleware Doctoral Symposium of the 13th ACM/IFIP/USENIX International Middleware Conference*. ACM, 2012.

[82] Masoomeh Rudafshani, Paul AS Ward, and Bernard Wong. MemRed: Towards reliable web applications. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*. ACM, 2012.

[83] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[84] Selenium. Selenium: Browser automation framework. https://code.google.com/p/selenium. Accessed Oct. 2014.

[85] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. ACM, 2013.

[86] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D Keromytis. Assure: Automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 37(1):37–48, 2009.

[87] StatCounter. StatCounter Global Stats. http://gs.statcounter.com/. [Accessed May 2012].

[88] Yusuke Suzuki. Escodegen. https://github.com/Constellation/escodegen. [Accessed October 2013].

[89] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM SIGOPS Operating Systems Review*, 37(5):207–222, 2003.

[90] Yan Tang, Qi Gao, and Feng Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX 2008 Annual Technical Conference*, pages 307–320. USENIX Association, 2008.

[91] The Meteor Project. An interesting kind of JavaScript memory leak. https://www.meteor.com/blog/2013/08/13/an-interesting-kind-of-javascript-memory-leak.

[92] The WebKit Open Source Project. SunSpider JavaScript benchmark. https://www.webkit.org/perf/sunspider/sunspider.html. [Accessed August 2014].

[93] TodoMVC. TodoMVC: Helping you select an MV* framework. Accessed Oct. 2014.

[94] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 49–59. ACM, 2014.

[95] Kalyanaraman Vaidyanathan and Kishor S. Trivedi. A comprehensive model for software rejuvenation. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):124–137, 2005.

[96] viterbiSearcher. Garbage collection. http://everything2.com/title/garbage+collection. [Accessed March 2015].

[97] W3C. Document Object Model (DOM) core level 2 specification. http://www.w3.org/TR/DOM-Level-2-Core/core.html. [Accessed March 2015].

[98] W3C. DOM level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/. [Accessed July 2014].

[99] The World Wide Web Consortium (W3C). HTML5. http://www.w3.org/TR/html5/. [Accessed March 2015].

[100] Gregor Wagner, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Compartmental memory management in a modern web browser. *ACM SIGPLAN Notices*, 46(11):119–128, 2011.

[101] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying Ye. *Probability and Statistics for Engineers and Scientists*, volume 5. Macmillan New York, 1993.

[102] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 473–486. USENIX Association, 2013.

[103] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346. ACM, 2013.

[104] Edmond Woychowsky. *Creating Web Pages with Asynchronous JavaScript and XML*. Prentice Hall, 2007.

[105] Andrew Wright. Garbage collection. http://www.cs.princeton.edu/courses/archive/spr96/cs441/notes/l13.html. [Accessed March 2015].

[106] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. *ACM SIGPLAN Notices*, 46(6):270–282, 2011.

[107] Guoqing Xu and Atanas Rountev. Precise memory leak detection for Java software using container profiling. In *30th International Conference on Software Engineering ICSE'08.*, pages 151–160. IEEE, 2008.

[108] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116. USENIX Association, 2006.

[109] Kevin Yank. How not to write JavaScript. http://www.sitepoint.com/google-closure-how-not-to-write-javascript/. [Accessed February 2015].